VERIFICATION OF SIMULATION MODELS OF NETWORK PROTOCOLS USING STATE SPACE EXPLORATION

BY

AHMED ADEL SOBEIH

B.Eng., Cairo University, Egypt, 1999
M.Eng., Cairo University, Egypt, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Associate Professor Mahesh Viswanathan, Chair
Assistant Professor Darko Marinov
Professor José Meseguer
Professor Klara Nahrstedt
Professor Lui Sha

Abstract

Verification and Validation (V&V) is a critically important phase in the development life cycle of a simulation model. In the context of network simulation, traditional network simulators perform well in using a simulation model for evaluating/predicting the performance of a network protocol but lack the capability of verifying the "correctness" of the simulation model being used. To address this problem, we have extended J-Sim — an open-source component-based network simulator written entirely in Java — with a *state space exploration* (SSE) capability that explores the (entire) state space created by a network simulation model in order to find an execution (if any) that violates an assertion; i.e., a property that must always hold true in all states.

In this thesis, we present the design and implementation of the state space exploration framework in J-Sim. Furthermore, we demonstrate its usefulness and effectiveness in verifying complicated simulation models. Specifically, we verify the simulation models of two widely used and fairly complex routing and data dissemination protocols: the *Ad-hoc On-Demand Distance Vector (AODV)* routing protocol for wireless ad-hoc networks and the *directed diffusion* data dissemination protocol for wireless sensor networks. Moreover, we verify the simulation model of a reliable unicast protocol: the *Automatic Repeat reQuest (ARQ)* protocol.

To enable the verification of these fairly complex network simulation models, we make use of protocol-specific properties along two orthogonal dimensions: *state similarity* and *state ranking*. State similarity determines whether a state is "similar to" another in order to enable the implementation of stateful search. On the other hand, state ranking determines whether a state is "better than" another with respect to the implementation of best-first search (BeFS). Specifically, we develop protocol-specific search heuristics to guide state space exploration towards finding assertion violations in less time. In particular, we report findings on how to devise *good* search heuristics for routing/data dissemination protocols similar to AODV and directed diffusion.

We evaluate the efficiency of our state space exploration framework by comparing its performance to that of a state-of-the-art model checker for Java programs, namely Java PathFinder (JPF). The

results of the comparison show that the time needed to find an assertion violation by our state space exploration framework in J-Sim can be significantly less than that in JPF unless a significant amount of programming effort is done in JPF to make its performance close to that of our SSE framework.

Finally, we present *incremental* state space exploration (ISSE), a technique that aims to provide a speedup in the state space exploration time of evolving simulation models; i.e., simulation models whose code changes from one version to another. A code change may or may not lead to a behavioral change. We analytically obtain necessary conditions for the ISSE technique to provide a speedup in the state space exploration time when compared to a traditional (non-incremental) SSE technique. We applied the ISSE technique to our case studies. In two case studies (namely AODV and directed diffusion), ISSE provided a speedup whereas in one case study (namely ARQ), it did not provide a speedup because the necessary conditions were violated.

To the memory of Prof. Jennifer Hou (d. 2007)

Acknowledgments

This thesis would not have been possible without the help, support and encouragement of many people. First of all, I would like to thank Prof. Jennifer C. Hou for being my PhD adviser until the last moments of her short life. Jennifer was a very dedicated adviser, an excellent researcher and an outstanding teacher. I also admire her for having a great devotion to serving the community through applying her research to meet people's needs and touch people's lives. I learned a lot from Jennifer during my stay at the University of Illinois. One thing that I especially learned from Jennifer is a quote that she held, firmly believed in and applied: "It's not whether you get knocked down. It's whether you get up again." I do plan to apply this quote in my life. Jennifer, you will always be remembered.

Second, I would like to thank Prof. Mahesh Viswanathan for being my PhD co-adviser and for taking over the leadership in advising me after Jennifer has passed away. I would like also to thank Prof. Darko Marinov for a lot of very useful comments and suggestions that helped improve the quality of this thesis. I also thank Darko for giving me the opportunity to work very closely with other graduate students in his research group during the past two years and for allowing me to run a lot of experiments on one of the machines in his group. I would like to thank Prof. José Meseguer and Prof. Lui Sha for serving on my PhD committee and for several useful comments and suggestions for improvements. I would like to thank Professor Klara Nahrstedt for her willingness to serve on my PhD committee. I would like to extend my thanks and gratitude to all the teachers and professors who taught me during the past twenty eight years.

In particular, I would like to thank Peter Ölveczky for a lot of help and useful comments.

I would like also to thank the following colleagues (in alphabetical order): Marcelo d'Amorim, Wei-Peng Chen, Jihyuk Choi, Yan Gao, Guanghui He, I-Hong Hou, Chunyu Hu, Joseph Jeon, Srikanth Kandula, Hwangnam Kim, LaeYoung Kim, Tae Hyun Kim, Tae Seok Kim, Lu-chuan Kung, Kiran Lakkaraju, Ray Kong Lam, Steven Lauterburg, Jong-Kwon Lee, Ning Li, Hyuk Lim, Ting-Yu Lin, Kyung-Joon Park, Deepu Thomas, Hung-Ying Tyan, Jun Wang, Yong Yang, Xiaoxin

Yin, Zheng Zeng, Honghai Zhang and Rong Zheng.

I am grateful to the Vodafone-U.S. Foundation for supporting me with the Vodafone-U.S. Foundation Graduate Fellowship for two consecutive years (2006-2007 and 2007-2008) and to Richard T. Cheng for establishing the Richard T. Cheng Fellowship that I received in the academic year 2002-2003. I would like to thank William (Bill) Yurcik for supporting me with a research assistantship at the National Center for Supercomputing Applications (NCSA) from July 2003 to August 2004. I would like also to thank Prof. Jennifer C. Hou and Prof. Mahesh Viswanathan for supporting me with research assistantships. I would like to thank IBM Research for giving me an IBM Invention Award in 2007.

I would like also to thank my parents who were my first instructors in this world. Since I was a child, they have been doing their best to help and encourage me towards success. Feeling especially indebted to them, I wish them all the best and do hope that one day, I will be able to equally reward them (if an equal reward is ever possible). I would like to extend my thanks to all the other members of my family and to all my friends, who are too many to mention, in Egypt, the United States, and Canada.

Finally, I would like to thank all fellow scientists and researchers.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AODV	Ad-hoc On-Demand Distance Vector routing
ARQ	Automatic Repeat reQuest protocol
BFS	Breadth-first Search
BFS-AC	Breadth-first Search, Add Current
BFS-ACS	Breadth-first Search, Add Current with Shuffle
BFS-AN	Breadth-first Search, Add Next
BFS-ANS	Breadth-first Search, Add Next with Shuffle
BeFS	Best-first Search
BeFS-AC	Best-first Search, Add Current
BeFS-ACS	Best-first Search, Add Current with Shuffle
BeFS-AN	Best-first Search, Add Next
BeFS-ANS	Best-first Search, Add Next with Shuffle
DFS	Depth-first Search
DFS-AC	Depth-first Search, Add Current
DFS-ACS	Depth-first Search, Add Current with Shuffle
DFS-AN	Depth-first Search, Add Next
DFS-ANS	Depth-first Search, Add Next with Shuffle
DFS-R	Depth-first Search, Recursive
DFS-RS	Depth-first Search, Recursive with Shuffle
ISSE	Incremental State Space Exploration
JPF	Java PathFinder
JVM	Java Virtual Machine
SSE	State Space Exploration

Chapter 1

Introduction

A *simulation* is the imitation of the operation of a real-world system over time [9]. The behavior of the system is studied by developing a *simulation model* that usually involves making a set of assumptions on the operation of the system. A "correct" simulation model can be extremely valuable for answering a wide variety of what-if questions about the behavior of the real-world system. In fact, a simulation model can be effectively used both as an analysis tool for explaining the behavior of, or predicting the effect of proposed change(s) to, an existing system and as a design tool for predicting the performance of a new system under various scenarios and refining the design of the system if necessary.

The applications of simulation vary widely and span a lot of civilian and military applications. Examples of the various applications of simulation include: computer networks and communications systems, operating systems, databases, transaction processing systems, manufacturing and material handling applications, transportation systems such as airports and freeways, construction engineering and project management, business processes, service organizations such as restaurants and post offices, inventory analysis, logistics and supply chain applications, health care, future combat systems (FCS), 3D military-based interactive simulation of a battlefield, training environments and flight simulators.

A thorough simulation study systematically consists of a set of steps (or phases). These steps include: problem formulation, identification of the alternative feasible solutions, setting the overall plan and the objectives of the study (e.g., selecting the solution with the highest benefits/cost ratio), construction of the model and collecting the required input data, implementing the simulation model using a programming language and/or simulation software, verification and validation of the model, design of the relevant experiments (also called simulation runs), experimentation and producing the simulation results, analysis of the output obtained from the simulation runs, and documenting the model, the development progress and the output. See [6,9] for detailed explanations of each of these steps.

1.1 Focus

One of the most difficult and important problems that a model developer has to encounter is the verification and validation (V&V) of the simulation model. A model is, by definition, an abstract representation of a real (existing or proposed) system built for the purpose of studying the system according to certain objectives. In particular, the *conceptual model* is a mathematical/logical/verbal representation of the system. In the conceptual model, we determine the assumptions on each system component, structural assumptions on the interactions between components, and data assumptions on the input data. In order to conduct simulation experiments on a model of a real-world system, the model is usually implemented using a programming language and/or simulation software on a computer; i.e., turning the conceptual model into a *computerized model*. Validation addresses the question of "Are we building the right model?" and verification addresses the question of "Are we building the model right?" [9]. In particular, verification focuses on the transformational accuracy when the model is transformed from one form into another [6].

The analysts who make use of the outputs of the simulation model to help in making important design recommendations and the managers who use these recommendations to make decisions (e.g., to compare between different systems or different designs of the same system) usually look upon a simulation model with some level of skepticism. After all, the model is just an abstraction. It is the job of the model developer(s) to work with the analysts and/or the managers to reduce this skepticism and enhance the objective credibility of the model. This makes verification and validation an important and integral part of the development life cycle of a simulation model. Computer-aided tools that can help the model developer(s) verify and validate their models thus become invaluable.

Some of the V&V methods are informal subjective comparisons between the output of the (computerized) model and the output of the real system; some are formal objective procedures based on statistical techniques such as hypothesis testing or confidence intervals. Examples of V&V techniques are:

1. Driving the simulation model with historical input data or samples generated from distributions and comparing the model input-output transformations to corresponding input-output transformations for the real system,

2. Face validity; i.e., domain experts and knowledgeable persons evaluate model output for reasonableness,

3. Checking the model assumptions (both structural assumptions on how the system operates

and data assumptions on the input data),

4. Sensitivity analysis; i.e., studying how the model output behavior is affected by systematically changing the values of the model input data over a certain range of interest,

5. Turing tests; i.e., domain experts and persons knowledgeable about system behavior are asked to compare, and determine if they can differentiate, between the model output and the system output under the same input conditions,

6. Using a simulation trace; i.e., a detailed computer printout that shows how the state of the simulation model changes over time,

7. Fault/failure insertion testing; i.e., inserting a fault (incorrect model component) or a failure (incorrect behavior of a model component) into the simulation model and testing whether the model generates the expected invalid behavior,

8. Regression testing; i.e., testing whether changes made to the simulation model have caused previously fixed errors to re-emerge,

9. Extreme or invalid input testing, and

10. Static and dynamic testing of the computerized model.

See [3–7, 58] for several verification and validation techniques at the different stages of the development life cycle of a simulation model.

Verification and validation is *not* an afterthought. Rather, the process of building, verifying and validating a simulation model is iterative and may have to be repeated several times over multiple versions of the simulation model until a certain level of credibility in the model (and the information derived from it) is achieved. Feedback and insights gained from the identification of the initial model deficiencies help improve the simulation model as it evolves over time.

In this thesis, we focus on the computer-aided verification of simulation models of network protocols with particular interest in evolving simulation models whose code changes from one version to another.

1.2 Motivation

The motivation of our work is that network simulators have been used for decades to build a model of a network protocol and evaluate its performance under different scenarios. One major deficiency

of traditional network simulators, however, is that they only evaluate/predict the performance of network protocols in the scenarios provided by the user (e.g., a network protocol designer or a simulation modeler) but can *not* exhaustively analyze all possible scenarios for correctness in terms of whether or not the simulation model satisfies certain assertions that the network protocol satisfies. For instance, a network simulator can use a simulation model of a routing protocol to conduct experiments and evaluate the performance of this protocol (e.g., in terms of the control message overhead), but cannot check whether the simulation model being used may suffer from routing loops.

If the assertion violations in the simulation model do not appear (and hence cannot be investigated) in the scenarios studied, they may not be identified as early as possible. If an assertion violation exists in the simulation model, the results obtained from the simulation experiments may be incorrect and may lead to wrong decision making. Furthermore, if the simulation model models a nonexistent system, undiscovered assertion violations in the simulation model may eventually manifest themselves only after the system has been implemented and deployed. In the light of recent research [85] that creates a physical implementation of a network protocol from existing simulation code without much modification, the consequence of leaving assertion violations undiscovered in the simulation code is especially severe.

Although there are hundreds of tools and utilities that are available in either the academic communities or the commercial markets and that can be used to provide computer-aided support for software testing, we agree with Balci et al. [7] that software tools *specifically* created for the verification of simulation models would provide much more effective computer-aided support. Design of special-purpose verification tools for simulation models of network protocols enjoys several benefits over using general-purpose verification tools for programming languages such as C/C++ or Java (e.g., [20, 25, 36, 111]). First, verifying the code of the simulation model *along* with the code of the simulation environment and its libraries might likely be intractable due to the complexity of the simulator libraries. Second, in many cases, the verification tool specifically designed for the simulation model of a network protocol can conveniently ignore the call stack and heap during the verification process, and this can lead to a more tractable analysis.

In the light of this motivation, we believe that building an *integrated* environment [95], which allows the user to *both* verify a simulation model of a network protocol *and* use it to evaluate/predict the performance of the protocol, is an important task.

1.3 Problems and Suggested Solutions

As indicated by Sargent in [86], there is no set of specific tests that can be easily applied to determine the "correctness" of a simulation model and that no algorithm exists to determine what techniques or procedures to use. As pointed out by Balci et al. in [7], computer-aided support for verification, validation, and accreditation (VV&A) is one of the strategic directions to achieve VV&A. In this thesis, we elaborate on a computer-aided technique that is complementary to existing verification and validation techniques and can be used in combination with them.

Specifically, we present the design, implementation and evaluation of a software framework that aids in the verification of a simulation model of a network protocol. Our framework facilitates this computer-aided support by providing the assertion checking V&V technique [6] *along several execution paths* in the state space created by the model. An assertion is a property that must always hold true (e.g., the absence of routing loops in the simulation model of a routing protocol).

We have implemented our framework in J-Sim [50, 88, 89, 108–110]—an open-source component-based network simulation environment written entirely in Java. Precisely, we have extended J-Sim with the state space exploration (SSE) capability that directly executes the simulation model, which is also written in Java, along several execution paths and systematically explores the (entire) state space created by the model up to a configurable maximum depth in order to check whether the model satisfies certain assertions that the network protocol satisfies and to find violations of these assertions if any exists. However, the challenge is how to enable the state space explorer to take control of the network simulation model to explore the (entire) state space, rather than simply exploring one single execution path (i.e., sequence of events) as J-Sim traditionally does. Moreover, this has to be done *without* requiring the core design and implementation of J-Sim to be altered and *without* degrading the execution time of the J-Sim simulation model if the user is only interested in using the model for performance evaluation purposes. In Chapter 3, we explain how we have overcome that challenge.

As mentioned above, simulation models do evolve over time. Consequently, the verification and validation process may have to be repeated several times over multiple versions of a model. Some code changes may lead to changes in the output of a model (e.g., bug fixes, eliminating model deficiencies, adding a new functionality to the model, removing an existing functionality from the model, or modifying the implementation of an existing functionality in the model). We call these code changes *behavioral changes*. On the other hand, some other code changes may not lead to changes in the output of a model (e.g., refactorings and code optimizations such as replacing a `for`

loop by a while loop). We call these code changes *non-behavioral changes*.

Evolution of a simulation model over multiple versions can pose a challenge for verification. A traditional state space explorer would take *one version* of the simulation model at a time and explore the (entire) state space created by each version *separately* to find assertion violations. This repeated verification may take an excessively long time. We notice, however, that evolution of a simulation model can also provide an opportunity for reducing the total verification time of all versions. Specifically, consecutive versions of a simulation model can have (significant) similarities that a traditional state space explorer does not take advantage of. In order to address this problem, we present *incremental* state space exploration (ISSE), a technique that aims to provide a speedup in the state space exploration time of evolving simulation models. Incremental state space exploration considers *several versions* of the simulation model and reuses the results of verifying one version to speed up the verification of a subsequent one. However, the challenge is to identify when an incremental state space explorer cannot provide a speedup in state space exploration (e.g., when the similarities between consecutive versions of a simulation model are not significant enough). In Chapter 7, we explain how we have overcome that challenge.

In the following section, we give a brief summary of our research achievements.

1.4 Research Summary and Contributions

This section summarizes the research activities and the contributions of this thesis.

1.4.1 State Space Exploration Framework

As a first step towards realizing our goals, we have implemented the state space exploration (SSE) framework in Java as a component in the component-based architecture of J-Sim for the purpose of seamless integration with J-Sim. We made use of J-Sim features, whenever possible, to enable the interaction between the state space explorer component and the simulation model of the network protocol in a way that enables the state space explorer to take control of the execution of the model, systematically explore the (entire) state space created by dynamically executing the model along several execution paths and report assertion violations if any exists. We implemented the SSE framework in a way that satisfies two major design goals: (i) the core design and implementation of J-Sim must not be modified, and (ii) any modifications to the J-Sim simulation model of the network protocol must be minimal and must not degrade the execution time of the model if a user

is only interested in using the model for performance evaluation purposes.

The SSE framework provides an explicit-state stateful search that can prune a path in the search space based on either the most recently visited state or its hash code. The SSE framework can employ several search strategies such as breadth-first (BFS), depth-first (DFS) and best-first (BeFS). To allow for variations in the search order across several executions of the SSE framework, we implement randomized versions of these search strategies. In Chapter 3, we provide the implementation details of our SSE framework and list what the user needs to do in order to verify the simulation model of a network protocol in J-Sim.

This contribution is unique because this is the *first* effort to integrate state space exploration with simulation in a unified J-Sim framework. This allows the user to *both* verify the simulation model *and* use it to evaluate the performance of a network protocol.

1.4.2 Case Studies

After the state space exploration framework is laid, we demonstrate the usefulness, effectiveness, and generality of our framework in verifying the simulation models of two widely used and fairly complex network protocols: the *Ad-hoc On-Demand Distance Vector (AODV)* routing protocol [77, 78] for wireless ad-hoc networks and the *directed diffusion* data dissemination protocol [48] for wireless sensor networks. These are reasonably complex network protocols whose J-Sim simulation models (not including the J-Sim library) have about 1200 and 1400 lines of code, respectively. (As a third case study [93], we also verify the simulation model of an *automatic repeat request (ARQ)* protocol, also known as the *alternating bit* protocol.) Our choice of AODV and directed diffusion is motivated by their potential to become representative routing and data dissemination protocols, respectively, in ad-hoc networks and sensor networks. We investigate whether these simulation models satisfy the *loop-free* assertion; i.e., data packets are not routed through loops.

Our discoveries [94], reported in detail in Chapters 4-5, illustrate the practical utility of our state space exploration framework. First, we find a previously unknown assertion violation in the J-Sim simulation model of AODV. This shows that even if the network protocol specification [77] is correct, the simulation model could violate some assertions that might eventually lead to inaccurate simulation results. Second, we identify previously unknown behaviors, which might happen in a wireless sensor network that is using the directed diffusion protocol. Specifically, our state space exploration framework produces scenarios leading to the corruption of data caches due to timeouts and/or node reboots in a sensor network. These scenarios would result in data packets being routed

in a loop. Third, we also demonstrate the ability of our state space exploration framework to find manually injected errors in the simulation models.

1.4.3 Protocol-specific Properties

One of the major challenges of state space exploration is the well-known *state space explosion* problem; i.e., the state space of the system can be so prohibitively large that a state space explorer may run out of memory. To handle this problem and enable the analysis of such large simulation models of real network protocols, we make use of algorithms and heuristics that exploit structural properties salient in the state space of message passing systems in general, and some network protocols in particular.

The first technique exploits the existence of a non-trivial *simulation relation* between states to reduce the search space. The idea is as follows. A traditional state space exploration tool performs a classical search algorithm (like depth-first or breadth-first search) on the directed graph of states and transitions defined by the protocol, choosing to prune certain branches of the search when it visits a state previously encountered. We observe that this basic algorithm is sound even when branches of the search are pruned whenever a state s is visited that can be *simulated by* another state s' that has been explored before, not just when the *same* state is re-encountered, i.e., whenever there is some previously visited state s' that can exhibit every behavior of s; the formal definition of simulation is deferred to technical sections (Section 3.3.1). This can drastically reduce the search space, if the simulation relation is good. Next, we observe that, when the communication between network entities is unreliable (i.e., when message delivery is neither guaranteed nor ordered), which is typically the case for end-to-end protocols for wireless networks, there is a very simple and natural simulation relation. Note that the state space of such protocols will consist of states comprising of a protocol state (encoding information needed in modeling the specific protocol) and an unordered collection of messages that have been sent but not yet delivered. For states s and s' with identical protocol states, and with the additional property that every undelivered message in s is also undelivered in s' (s' could have more undelivered messages), we observe that s' simulates s. We exploit this simulation relation in our experiments in Chapters 4-5. In Appendix A, we argue that each of the simulation relations that we used is indeed a simulation relation.

Our second technique, called *state ranking*, is used to direct the state space exploration more effectively towards finding assertion violations quickly. State ranking, as the name suggests, orders states based on which state is more "likely" to have an incorrect execution starting from it. The state

space exploration framework then uses a *best-first search (BeFS)* algorithm that biases the search towards states with higher ranks; more specifically, the algorithm after visiting a state, always picks successor states to visit in the order of their ranks. To obtain a ranking of states, we exploit *properties inherent to the network protocol and the assertion being checked*. However, unlike the simulation relations that we exploit, the state ranking that we present for the examples is not provably correct. In other words, we cannot statically prove that states with higher ranks in our schemes have more incorrect executions. Best-first search, though a heuristic, does not affect the soundness of our search engine (since it only changes the order of the search) and can find assertion violations several orders of magnitude faster than classic breadth-first and depth-first search strategies, as our experiments demonstrate.

One interesting and important research question is how to determine a suitable BeFS heuristic for a specific network protocol. We make an attempt towards answering this question by studying the performance of several BeFS heuristics for both AODV and directed diffusion, and providing design guidelines based on our results.

1.4.4 Comparison with the State of the Art

We evaluate the efficiency of our state space exploration framework in J-Sim by comparing its performance to that of a state-of-the-art model checker for Java programs, namely Java PathFinder (JPF) [52, 111]. JPF is implemented in Java as a special Java Virtual Machine (JVM) that runs on top of a host JVM. The main difference between JPF and a regular JVM is that JPF implements a *backtrackable* JVM; i.e., JPF can (quickly) backtrack the program execution by restoring a state that was previously encountered during the execution. Backtracking allows exploration of different executions from the same state.

We compare between our SSE framework in J-Sim and that in JPF in the cases where the explicit-state stateful search prunes a path in the search space based on either the most recently visited state or its hash code. In order to ensure a fair comparison between the two tools, we used the same definition of the simulation relation that enables the stateful search, the same implementations of BeFS heuristics for state ranking, and the same hash code computation function in both J-Sim and JPF. This ensures that the number of transitions executed and the number of states visited is the same in both tools.

The results of the comparison [90], reported in detail in Chapter 4, show that our SSE framework can be significantly faster than JPF in terms of the time needed to find an assertion violation unless

a significant amount of programming effort is done in JPF to make its performance close to that of our SSE framework. This result justifies the need for building a framework that is specifically created for the verification of network simulation models instead of using a general-purpose model checking tool since we believe that network protocol designers and simulation modelers will feel more comfortable using J-Sim as a single integrated environment for both building a simulation model and verifying its correctness than using J-Sim for building a simulation model and using another tool (JPF) for verifying its correctness.

To the best of our knowledge, this is the first research effort to compare JPF with a state space explorer for the simulation models of network protocols.

1.4.5 Incremental State Space Exploration Framework

In Chapter 7, we provide the implementation details of the incremental state space exploration (ISSE) framework that aims to speed up the verification time of evolving simulation models in J-Sim. The basic idea is to make use of the results of verifying one version of the model to speed up the verification of a subsequent one by avoiding redoing certain operations such as re-executing certain transitions and/or re-checking the assertions and recomputing the hash codes in certain states. We implemented the ISSE technique in J-Sim as a general case of one of the (non-incremental) SSE procedures in our SSE framework (Section 1.4.1). Specifically, the non-incremental SSE procedure is just one mode of four possible modes of operation of ISSE. This generalization enables applying the ISSE idea to other explicit-state model checkers in a straightforward manner. In fact, our work [92] on ISSE in J-Sim is a part of a larger work [57] on incremental state space exploration for model checking programs with dynamically allocated data, including the implementation and evaluation of the ISSE technique in Java PathFinder (JPF).

An interesting research question is whether or not the ISSE technique can backfire (i.e., make the time needed for verifying a model using ISSE longer than that using the traditional, non-incremental SSE technique). In order to identify when the ISSE technique cannot provide a speedup in state space exploration of several versions of the simulation model, we analytically obtain necessary conditions for the ISSE technique to provide a speedup in the state space exploration time when compared to the non-incremental SSE technique. The necessary conditions depend on (i) the degree of the similarities between successive versions of the simulation model, (ii) properties related to the state space created by the model, and (iii) the times spent in the different operations during state space exploration in our framework. Our implementation dynamically checks whether or not these necessary conditions

are satisfied. Hence, the user may run a pilot study to determine whether the ISSE technique backfires.

In Chapter 8, we apply the ISSE technique to our case studies and compare its performance with that of the non-incremental SSE technique. To ensure a fair comparison in all our experiments, we check that the state spaces explored by both techniques are the same; i.e., both techniques have the same "workload". We consider both behavioral and non-behavioral changes. In two case studies (namely AODV and directed diffusion), ISSE provided a speedup whereas in one case study (namely ARQ), it did not provide a speedup because the necessary conditions were violated. The speedups in state space exploration time come at a reasonable cost of memory overhead.

1.5 Thesis Organization

After this introduction, the rest of the thesis is organized as follows. In Chapter 2, we provide background information and give a brief overview of network simulation in J-Sim. In Chapter 3, we elaborate on the state space exploration (SSE) framework that we implemented in J-Sim. Chapters 4-6 present the case studies and the performance results using the SSE framework. In Chapter 7, we elaborate on the incremental state space exploration (ISSE) framework that we also implemented in J-Sim. Chapter 8 presents the performance results using the ISSE technique. In Chapter 9, we discuss related work. Finally, we conclude the thesis in Chapter 10 with a list of potential future research work.

Chapter 2

Background

In this chapter, we provide background information. Section 2.1 defines some basic concepts. Section 2.2 gives a brief overview of formal reasoning in general and explicit-state model checking by state space exploration in particular. Finally, Section 2.3 explains network simulation in J-Sim.

2.1 Basic Concepts

First, we review some basic concepts of discrete-event system simulation [9].

A *system* is a group of objects that are joined together in some regular interaction or interdependence toward the accomplishment of some purpose (e.g., a wireless network). A *simulation model* is a representation of the system for the purpose of studying the behavior of the system with respect to certain performance evaluation criteria (e.g., network throughput). An *entity* is an object of interest in the system (e.g., a wireless node). An entity is described or characterized by its *attributes* (e.g., routing table, MAC address, etc.). Each attribute has a type, which may be either simple (e.g., boolean, integer, etc.) or composite (e.g., an array of integers, an object, an array of objects, etc.). The type of an attribute indicates the domain over which the attribute ranges.

The *state* of the system is a complete description of the system and includes values of all attributes of entities that are relevant to the objective of the study (e.g., the routing table entries of all the wireless nodes and the number/contents of the messages being sent over the wireless channel). A state s can be regarded as a function that assigns to each attribute a value over its domain. The *state space*, denoted by S, is the set of all possible system states. It should be noted that although the set of attributes is finite, S may be infinite (or prohibitively large) because the domains over which the attributes range may be infinite (or prohibitively large).

An *event* is an instantaneous occurrence that might change the state of the system (e.g., message sending and message receiving); i.e., assign new values to (some of) the attributes. An event may be either conditional or unconditional. An *unconditional event* is an event that can always occur

(e.g., an unpredictable crash of an active wireless node). A *conditional event* is an event that can occur only if a certain *enabling condition* is true (e.g., a wireless node can receive a message only if another node has sent a message to it).

An *assertion* is a property that must always hold true in all states (e.g., the absence of routing loops in a routing table). The meaning of the assertion depends on the network protocol itself. For example, if the protocol is a reliable unicast/multicast protocol, the assertion may be that the receiver(s) receive all the packets that the sender believes to have been received. In the case of a security protocol, the assertion may be that unauthorized users do not get access to the system.

2.2 Formal Reasoning and State Space Exploration

Next, we give a brief overview of formal reasoning in general and explicit-state model checking [21] by state space exploration in particular.

In general, there are two basic approaches towards formal reasoning of software and hardware systems: *theorem proving* and *model checking*. In theorem proving, a formal technique (e.g., deductive methods and induction) is used to prove a property of a system. On the other hand, model checking checks a finite state machine model of the system in order to verify whether a temporal property (e.g., safety or liveness) holds. In the finite state machine model, vertices correspond to states of the system and edges or transitions correspond to events that might change the state of the system.

In particular, explicit-state model checking [21] by state space exploration starts from an initial state of the system and recursively generates successor system states by executing the transitions of the system. This process continues, usually while pruning the search when a previously visited state is encountered, until either the entire state space is explored or a property violation is discovered. (Hence, in systems of infinite state space, model checking by state space exploration is practically used for locating property violations, rather than proving correctness.)

As compared to theorem proving, model checking by state space exploration has several important advantages. First, it can be built into existing tools and automated. Second, it does not require a deep understanding of complex mathematical concepts; hence, a network developer or a computer engineer can find it easy to use. Third, when the desired property fails to hold, a model checker provides a counterexample; i.e., a trace of the sequence of events that starts from the initial state and leads to the property violation. This counterexample usually helps in understanding why the

property violation occurred and how it can be fixed. For these reasons, we choose to incorporate explicit-state model checking by state space exploration in J-Sim. It should also be mentioned that our state space exploration framework checks assertions only. In our framework, the user specifies the assertion as a Java method whose output is true/false.

However, one of the major challenges of model checking is the well-known *state space explosion* problem, i.e., the state space of the system can be so prohibitively large that a model checker may run out of memory. Several approaches to handling the state space explosion problem (e.g., partial order reduction, symmetry, abstraction, just to name a few) can be found in [21]. In Chapter 9, we give a brief overview of some of these techniques.

Before explaining the design and implementation of our SSE framework in J-Sim, it is mandatory to first give a brief overview of network simulation in J-Sim.

2.3 Network Simulation in J-Sim

Modern data communication networks are extremely complex and do not lend well to theoretical analysis. With computer/network entities and techniques interacting and interfering with one another, theoretical network analysis can be rigorously made only after leaving out several (sometimes subtle) details that cannot be easily captured in the mathematical analysis [19, 65, 75, 87]. As a result, it may be more feasible to carry out simulation to study and evaluate the performance of network entities and protocols, and interaction among them. Several existing network simulators have been used for decades by network protocol designers, researchers and modelers to build simulation models of network protocols, evaluate their performance with respect to pre-selected networking metrics (e.g., system throughput, packet delivery ratio, and end-to-end delay) and re-design/refine them if needed.

Most notable research efforts on network simulation include: NEtwork Simulation Testbed (*NEST*) [28], The REalistic And Large (*REAL*) [54], ns-2 (ns version 2) [103], SSFNet [105], GloMoSim [101], OPNET [104] and J-Sim [50, 108]. In this section, we explain network simulation in J-Sim.

2.3.1 J-Sim Software Architecture

As indicated by Balci et al. in [7], developing modeling and simulation applications using the component-based technology [18] is one of the strategic directions to achieve VV&A. J-Sim [50, 108]

is an open-source network simulation and emulation environment that is developed entirely in Java on top of a component-based software architecture, called the *autonomous component architecture (ACA)* [108], that closely mimics the integrated circuit (IC) design. The basic entities in the ACA are *components*, which communicate with one another via sending/receiving data at their *ports*. When data arrives at a port of a component, the component processes the data immediately in an independent execution context (e.g., *thread* in Java).

We believe that the reason software design cannot achieve the same level of modularity as IC design is because the object-oriented (OO) programming paradigm is fundamentally different from hardware design in *component binding*. Specifically, in OO programming, a class makes direct references to other class instances and makes function calls to those exposed by other class instances. This implies that the binding is too strong in the sense that the caller has to know the exact names of the callees. Because of that, it is difficult to develop and maintain an OO software system with a large collection of functions and classes. In the course of debugging, one cannot obtain a clear view of binding relations without delving into the implementation details and tracing codes line by line. This yields unpredictability in software development and high maintenance cost, and is usually termed as software crisis.

On the other hand, in IC design, an IC chip is a blackbox fully specified by the function specification and the input/output signal patterns in the databook. Changes in input signals trigger an IC chip to perform certain function, and change, after a certain delay, its outputs according to the chip specification. The fact that an IC chip is interfaced with other chips/modules/systems only through its pins (and is otherwise shielded from the rest of the world) allows IC chips to be designed, implemented, and tested, independent of everything else. In other words, at design time, an IC chip is bound with a certain specification in the databook, instead of being bound to components that interact with it. Component binding is thus deferred to the time when a system (e.g., ALU) is being composed.

Following the same line of design principles of IC chips, how components in the ACA behave (in terms of how a component handles and responds to data that arrives at a port) is specified at the system design time in *contracts*, but component binding does not take place until the system integration time when the system (e.g., a simulation model of a computer network) is being "composed." A contract specifies how an initiator (caller) and a reactor (callee) fulfill a certain function; i.e., the causality of information exchange between components but *not* the components that may participate in information exchange. Two components, acting respectively as the initiator and the

reactor, are bound at the system integration time to fulfill the contract. A system with all the components bound to one another is said to be complete, if the initiators of all involved contracts are fulfilled. In some sense, the ACA realizes the notion of *software IC* [10] where an IC corresponds to a component, pins correspond to ports, signals correspond to data that arrives at a port of a component, and an IC specification corresponds to a contract.

2.3.2 J-Sim Features

With the separation of contract binding (at system design time) from component binding (at system integration time), J-Sim provides a loosely-coupled component architecture, i.e., a component can be individually designed, implemented and tested independently [50, 108]. By closing the gap between hardware and software ICs, the ACA realizes the objectives of composability [110]. Similar to composing an electronic system (e.g., ALU) by interconnecting ICs via pins, building a simulation model of a network protocol in J-Sim requires designing and implementing a set of components (e.g., senders, receivers, routers, links, and protocols that run within each router/host) and interconnecting them via ports. Furthermore, the ACA realizes the objectives of extensibility and loose coupling between individual components [108], which enable new components (e.g., a state space explorer) to be included into J-Sim in a plug-and-play fashion.

On top of the ACA, a generalized packet-switched internetworking framework (called *INET*) has been laid based on common features extracted from the various layers in the network protocol stack. Both the ACA and the INET have been implemented in Java, and the resulting code, along with its scripting framework and GUI interfaces, is called J-Sim. Finally, an essential suite of wired and wireless network components and protocols have been implemented in J-Sim. In [88, 89], we have implemented, and evaluated the performance of, a simulation framework for wireless sensor networks (WSNs) in J-Sim. Cross-layered approaches can be realized using the autonomous component architecture by appropriately connecting the ports of a protocol class to those of another protocol class (not immediately above/beneath the former protocol class). This allows information to be used (and decision to be made) in a cross-layered manner.

J-Sim possesses several other desirable features. The fact that J-Sim is implemented in Java, along with its ACA, makes J-Sim a truly platform-independent simulation environment. J-Sim provides a script interface that allows its integration with different script languages such as Perl, Tcl, or Python. In particular, the latest release of J-Sim (version 1.3) has been fully integrated with a Java implementation of Tcl interpreter, called Jacl, with the Tcl/Java extension. Therefore,

similar to ns-2 (ns version 2) [103], J-Sim is a dual-language simulation environment in which classes are written in Java (for ns-2, in C++) and "wired" together using Tcl/Java[1]. However, unlike ns-2, classes/methods/fields in Java need not be explicitly exported in order to be accessed in the Tcl environment. Instead, all the public classes/methods/fields in Java can be accessed (naturally) in the Tcl environment.

Interested readers are referred to [108] for a detailed qualitative comparison between J-Sim and other network environments (such as ns-2 [103] and SSFNet [105]). Furthermore, we have conducted a detailed quantitative comparative study [109] between J-Sim, ns-2 and SSFNet. The results of our empirical study indicate that J-Sim demonstrates better scalability because of better memory usage. In large-scale simulation scenarios (e.g., the number of nodes ≥ 18000), J-Sim outperforms ns-2 and SSFNET in terms of the simulation setup time and the simulation completion time. In small-scale simulation scenarios, the simulation completion time in J-Sim is no more than four times larger than that in ns-2.

For all the reasons mentioned above, we believe that J-Sim is a promising candidate for the network simulator to be extended with the state space exploration capability. However, we believe that our technique can be implemented in other network simulators too. Furthermore, although we propose our technique in the context of simulation of computer networks, the idea itself is generic enough and can be applied to other application domains of simulation.

[1]The term "wire" is the term that J-Sim uses when displaying a connection between two ports.

Chapter 3

State Space Exploration Framework

In this chapter, we elaborate on the state space exploration framework that we implemented in J-Sim. In Section 3.1, we describe our design goals. In Section 3.2, we explain the verification model. Section 3.3 gives a detailed explanation of the state space exploration process. Following that, Section 3.4 mentions the implementation problems that we faced and how we solved them. Finally, Section 3.5 summarizes what a user needs to do in order to verify the simulation model of a network protocol.

3.1 Design Goals

While building the state space exploration framework in J-Sim, we had two major design goals in mind:

1. The core design and implementation of J-Sim must not be modified.

2. Any modifications to the J-Sim simulation model of the network protocol must be minimal and must not degrade the execution time of the J-Sim simulation model if a user is only interested in using the model for performance evaluation purposes.

We believe that these two design goals should be also kept in mind if one desires to build a state space exploration framework in any other (network) simulator.

Towards realizing the above design goals, we implement an explicit-state stateful state space explorer in Java as a component in the ACA of J-Sim. The state space explorer executes the J-Sim simulation model of the network protocol *directly* and explores the state space on-the-fly. Specifically, the state space explorer starts from an initial state and generates successor states by executing the events of the simulation model. This process continues until either a counterexample disproving the assertion is found or the state space is explored up to a configurable maximum depth ($MAXDEPTH$). In the former case, the state space explorer outputs a counterexample; i.e., the

sequence of events that starts from the initial state and leads to violating the assertion. In the latter case, the state space explorer reports a message stating that "No assertion violation was found"; this does not mean that the simulation model does not violate the assertion and further testing is not required. A violation of the assertion may exist at depths larger than $MAXDEPTH$. Furthermore, the state space explorer may run out of memory (or take an excessively long amount of time) before exploring the state space up to $MAXDEPTH$. Hence, the goal of our framework is *not* to prove that the simulation model satisfies the assertions. Instead, the goal is to find violations of those assertions if any exists given a certain budget of time and memory constraints.

3.2 The Verification Model

For the purpose of performance evaluation, a simulation model contains only components of the real system that are relevant to measuring performance. Similarly, for the purpose of verification using state space exploration, a *verification model* contains only components of the simulation model that are relevant to state space exploration. In other words, we can ignore some attributes in the simulation model for verification purposes and get a model which is bisimilar to the simulation model. Formally, let L be the set of states in the simulation model, S be the set of states in the verification model, and $\alpha : L \longmapsto S$ be the mapping from L to S such that for any two states $l_1, l_2 \in L$, if $\alpha(l_1) = \alpha(l_2)$ then both $\alpha(l_1)$ and $\alpha(l_2)$ satisfy the same assertions. Now consider the states $l_1, l_2 \in L$ such that $\alpha(l_1) = \alpha(l_2)$. Let $l_1 \longrightarrow l_1'$ denote executing an event in the simulation model from state l_1 to state l_1'. The function α witnesses the bisimilarity of the simulation and verification models only if the following conditions hold: (1) If $l_1 \longrightarrow l_1'$, then $\exists l_2'$ such that $l_2 \longrightarrow l_2'$ and $\alpha(l_1') = \alpha(l_2')$, and (2) If $l_2 \longrightarrow l_2'$, then $\exists l_1'$ such that $l_1 \longrightarrow l_1'$ and $\alpha(l_1') = \alpha(l_2')$.

Based on the formal conditions in the previous paragraph, one can see that there are two criteria for one to ignore an attribute in the simulation model. First, forgetting it should result in no "change of behavior". Next, the attribute should not play a role in the checking of the assertion. An example of an attribute that is part of the simulation model but not part of the verification model is a fixed configuration parameter whose value does not change from one state to another (e.g., the maximum number of times to try sending a packet). Another example is an ACA timer. Since we do not consider the actual expiration times of a timer (see Section 3.4) and none of the assertions considered in this thesis depend on time, the expiration time of an ACA timer is not included in the verification model because it is irrelevant for state space exploration.

Hence, the definition of the verification model may change if the assertion that is to be checked changes, and it is the responsibility of the user to define the verification model and the mapping α mentioned above. This is similar to changing the simulation model if the objectives of a performance evaluation study change. Note that the verification model is *not* needed for measuring performance; it is only needed for checking whether or not the simulation model satisfies the assertion. In contrast, the simulation model *is* needed for state space exploration because the events are executed *inside* the simulation model. This will be explained in more detail in Section 3.3.2.

3.2.1 Global States

Figure 3.1 illustrates the overall framework of incorporating state space exploration into J-Sim. As shown in Figure 3.1, the state space explorer component interacts via ports with the protocol entities P_1, P_2, ... , P_n, which are instances of the Java classes that implement the simulation model of the network protocol. In order to explore the state space created by the simulation model, the notion of the "state" has to be adequately defined in the verification model. To this end, the state space explorer makes use of a class called *GlobalState*. The *GlobalState* class includes (some of) the attributes of the entities as data members. In the verification model, a state is an instance of the *GlobalState* class that assigns to each data member (i.e., attribute) a value. The implementation of *GlobalState* differs from one network protocol to another; hence, it is the responsibility of the user to provide an implementation of *GlobalState*. In addition, the user should also construct an initial state in order for state space exploration to get started.

As shown in Figure 3.1, the state space explorer interacts via ports with three instances of *GlobalState*, namely *initialState* (the initial state), *currentState* (the current state being explored) and *nextState* (one of the possible successors of the current state). The contract needed to define the information exchange between the state space explorer and the three instances of *GlobalState* is implemented in the *ModelCheckingGlobalStateContract* class, which is a subclass of *Contract*, a key J-Sim class that defines a contract. As mentioned in Section 2.3, a contract specifies the causality of information exchange between components but not the components that may participate in the information exchange. At the system design time, neither the initiator nor the reactor knows the identity of the other. The connection between the initiator and the reactor (which is shown using double-arrows in Figure 3.1) takes place only at the system integration time when the two components are bound to fulfill the contract. This ensures a loosely-coupled component architecture. Similarly, the *ModelCheckingProtocolEntityContract* class implements the contract needed to define

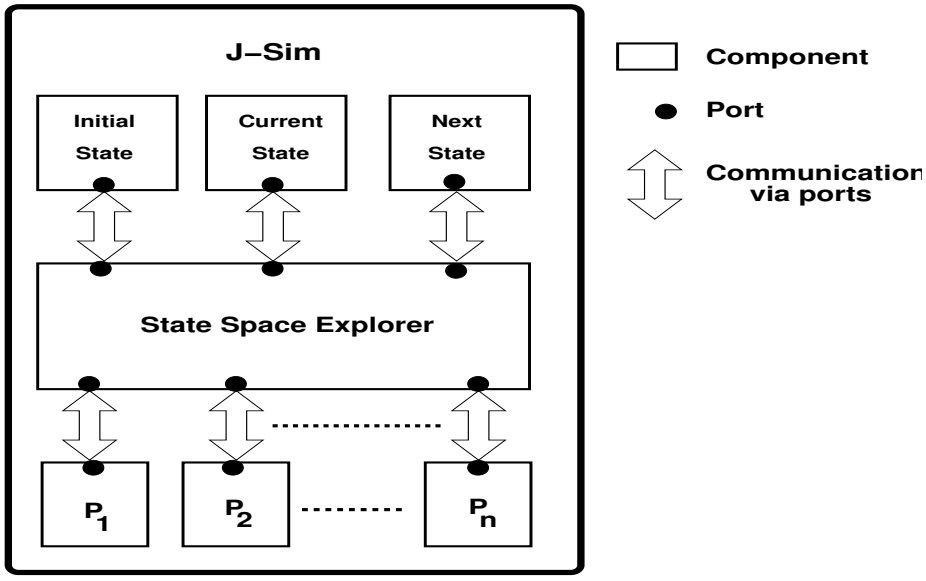


Figure 3.1: Overall framework of state space exploration in J-Sim. The protocol entities P_1, P_2, ... , P_n constitute the simulation model whereas the initial state, current state and next state constitute the verification model.

the information exchange between the state space explorer and the protocol entities.

Making the state space explorer take control of the simulation model in order to explore the (entire) state space, rather than just exploring one single execution path as J-Sim traditionally does, is achieved by having the state space explorer be the initiator of the *ModelCheckingProtocolEntityContract* (*ModelCheckingGlobalStateContract*) contract, and having a protocol entity (global state) be the reactor respectively.

3.2.2 Events

In each state in the state space, some events may or may not be enabled. Examples of events that are common among various network protocols are: packet reception, packet loss and timeout. Enabled events are parameterized; hence, an enabled event may generate multiple successor states depending on the values of its parameters. For instance, a packet reception event may generate multiple successor states because if the network contains K packets $m_1, m_2, \ldots, m_K$ whose destination is node n and the network does not guarantee ordered packet delivery, K successor states can be generated depending on which of the K packets is to be received by node n. On the other hand, a node reboot event may generate only one successor state (namely the state of the node after reboot). In the current implementation, the parameters of the events are limited to integer values.

For example, in the packet reception event described above, the index i ($0 \leq i \leq K-1$) of a packet in the network is used as a parameter to determine which packet is to be received.

It is important to note that the number of possible successor states that an enabled event can generate is state-dependent. In particular, an event may be enabled in one state but disabled in another. It is the responsibility of the user to specify (a) the finite set of events that can occur, (b) a function that dynamically determines the state-dependent number of possible successor states that each event generates (zero if the event is disabled); we call this function the *enabling function*, and (c) how each event is handled (i.e., an event handler function that makes a state change). Note that the enabling function is part of the verification model whereas the event handlers are part of the simulation model. This is because the user has to write the event handlers anyway in order to have a working simulation model of the network protocol in J-Sim, even if he/she does not intend to make use of the state space exploration framework for the purpose of verification.

3.2.3 State Space Graph

Although all the J-Sim simulation models are written in Java code and *not* in any formal definition language, it is useful to formally define the state space graph created by the simulation model of a network protocol, which we denote as $[\![P]\!]$. After defining the global states (Section 3.2.1) and the events (Section 3.2.2), $[\![P]\!]$ can be formally defined as the tuple $(\Sigma, S, \rightarrow)$ where Σ is the finite alphabet denoting the finite set of events that can occur, S is the set of global states, and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.

If $s, s' \in S$ and $e \in \Sigma$, then we write $(s, e, s') \in \rightarrow$ as $s \xrightarrow{e} s'$. To account for the fact that events are parameterized, we introduce the following extra notation. For a pair of states $s, s' \in S$ and i is an integer parameter of an event e such that $s \xrightarrow{e} s'$, we represent this transition as $s \xrightarrow{e(i)} s'$. In other words, $s \xrightarrow{e} s' \iff \exists i, s \xrightarrow{e(i)} s'$.

Let ϕ be an assertion. We write $s \vDash \phi$ to denote that the state $s \in S$ satisfies ϕ.

3.3 The State Space Exploration Process

In this section, we give a detailed explanation of the state space exploration process.

3.3.1 Stateful Search

Figure 3.2 shows the pseudo-code of the state space exploration procedure SSExploreAddNext(), which is one of the main procedures in the state space explorer component (Figure 3.1). The two major data structures in SSExploreAddNext() are *ToBeExplored* and *AlreadyVisited*. *ToBeExplored* stores the states from which no event has been explored yet. *AlreadyVisited* stores the states, or the hash codes of the states, that have already been visited.

If *AlreadyVisited* stores the states that have already been visited, a traditional explicit-state state space explorer would avoid (re-)exploring a state s_1 if the *same, identical* state were previously visited, i.e., were in the data structure *AlreadyVisited*. Figure 3.2, on the other hand, presents a modified explicit-state stateful search that avoids (re-)exploring a state s_1 if another state s_2 has already been visited before, which (provably) *simulates* it. Informally, s_1 is simulated by s_2 (or s_2 simulates s_1) if the states that can be explored from s_1 are also simulated by those that can be explored from s_2; hence, there is no need to explore states from s_1. Formally, a simulation relation is a binary relation R over the set of states S (i.e., $R \subseteq S \times S$) such that for every pair of states $s_1, s_2 \in S$, if $(s_1, s_2) \in R$ then (1) for every assertion ϕ, $s_1 \vDash \phi \iff s_2 \vDash \phi$, and (2) for each event e that is enabled in s_1, $s_1 \xrightarrow{e} s_1'$ where $s_1' \in S$ implies that e is also enabled in s_2, and $s_2 \xrightarrow{e} s_2'$ where $s_2' \in S$ and $(s_1', s_2') \in R$. In this case, we say that s_2 simulates s_1 (or s_1 is simulated by s_2). We also say that s_1 and s_2 are similar. Note that the simulation relation is a preorder; hence, it is reflexive and transitive.

As part of the verification model, the user writes a function $isSimulatedBy()$ such that a state s_1 is said to be simulated by another state s_2 if $s_1.isSimulatedBy(s_2)$ returns true. When the communication between network entities is unreliable (i.e., when message delivery is neither guaranteed nor ordered), which is typically the case for end-to-end protocols for wireless networks, there is a very simple and natural simulation relation. Note that the state space of such protocols will consist of states comprising of a protocol state (encoding information needed in modeling the specific protocol) and an unordered collection of messages that have been sent but not delivered. For states s and s' with identical protocol states, and with the additional property that every undelivered message in s is also undelivered in s' (s' could have more undelivered messages), we observe that s' simulates s. For the examples in Chapters 4-5 we exploit this simulation relation. However, in general, a user could potentially exploit specific details of the protocol to reveal a stronger simulation relation (and hence a faster exploration of a larger state space). Hence, we leave the implementation of $isSimulatedBy()$ to the user.

If *AlreadyVisited* stores the *hash codes* of the states that have already been visited instead of the states themselves, then the user writes a function $computeHashCode()$ that computes the hash code of a state as part of the verification model. The state space exploration procedure would then avoid (re-)exploring a state s_1 if another state s_2, having the same hash code as s_1, has already been visited before; i.e., $s_1.computeHashCode()$ exists in *AlreadyVisited*. The state space exploration procedure in this case is different from, but a straightforward modification of, SSExploreAddNext() (Figure 3.1) and we defer its detailed description and analysis to Chapter 7. In this chapter, we focus on the case where *AlreadyVisited* stores the states themselves.

Regardless of whether *AlreadyVisited* stores the hash codes of states or the states themselves, the goal of the stateful search in both cases is to turn the state space, which is usually a graph, into a finite *computation tree* [21]. The root of this tree is the initial state. The depth of a node (i.e., state) s is the length of the path from the root to s. The tree is finite for two reasons: (1) the state space is explored up to $MAXDEPTH$, and (2) the search is stateful.

3.3.2 Interaction with the State Space Explorer Component

SSExploreAddNext() starts with each of *AlreadyVisited* and *ToBeExplored* initially contains the initial state only (Figure 3.2, lines 3-4). As long as *ToBeExplored* is not empty (Figure 3.2, line 5), SSExploreAddNext() removes a state from *ToBeExplored* and sets *currentState* to it (Figure 3.2, line 6). This operation is an example of information exchange that makes use of the *ModelChecking-GlobalStateContract* contract. Specifically, the state space explorer is the initiator and *currentState* is the reactor. The state space explorer sends *currentState* the state s that has been removed from *ToBeExplored*. *currentState* responds by setting its attributes to the values assigned to them in the state s.

For each state being explored (*currentState*), SSExploreAddNext() first determines the events that are enabled in *currentState* by invoking *GenerateEnabledEvents()* (Figure 3.2, line 7). In *GenerateEnabledEvents()*, the enabling function (Figure 3.2, line 27) returns the number of possible successor states for each event (zero if the event is disabled). This operation is another example of an information exchange that makes use of the *ModelCheckingGlobalStateContract* contract because calculating the number of possible successor states is done inside the verification model. *GenerateEnabledEvents()* returns *EnabledEvents*, which is a list of enabled events (Figure 3.2, line 30). Each entry in *EnabledEvents* stores the corresponding event information *EventInfo* (Figure 3.2, line 29). Specifically, let each protocol entity have a unique ID p (Figure 3.2, line 25), each event have

a unique ID e (Figure 3.2, line 26), and each enabled event has a set of integer-valued parameters i where $0 \leq i \leq NumberOfNextStates - 1$ (Figure 3.2, line 28), then each instance of *EventInfo* stores p, e and i.

SSExploreAddNext() then generates each of the successor states (*nextState*) by calling the *GenerateNextState()* function (Figure 3.2, line 10) for each enabled event, which in turn invokes the corresponding event handler (Figure 3.2, line 34). Recall that the verification model is bisimilar to the simulation model and obtained by ignoring attributes that are either fixed or irrelevant. What this means is that if we instantiate the irrelevant variables with any values, execute the event inside the simulation model and then ignore the irrelevant/fixed attributes, we would be faithfully executing a single step of the verification model. Hence, an event handler is only invoked from the state space explorer but actually executed *inside* the simulation model; namely the protocol entities themselves. Therefore, SSExploreAddNext() must first set the state of the protocol entities to the state reflected in *currentState* before the execution of the event handler. This is achieved by the *CopyFromVModelToSModel()* function call (Figure 3.2, line 32). *CopyFromVModelToSModel()* is an example of an operation that makes use of both the *ModelCheckingGlobalStateContract* and the *ModelCheckingProtocolEntityContract* contracts. Specifically, the state space explorer uses *ModelCheckingGlobalStateContract* to query *currentState*, and *currentState* responds with the state of the protocol entities reflected in it. Following that, the state space explorer uses *ModelCheckingProtocolEntityContract* to instruct the protocol entities to set their state to the state reflected in *currentState*. Executing an event handler (Figure 3.2, line 34) is an example of an operation that makes use of the *ModelCheckingProtocolEntityContract* contract, where the state space explorer instructs a protocol entity to execute an event. After the execution of the event handler, the *CopyFromSModelToVModel()* function is called (Figure 3.2, line 35) to perform the reverse operation; i.e., extract the new state information from the protocol entities and copy them to *nextState*. *CopyFromSModelToVModel()* is another example of an operation that makes use of both the *ModelCheckingProtocolEntityContract* and the *ModelCheckingGlobalStateContract* contracts. (Note that *CopyFromSModelToVModel()* implements the mapping α mentioned in Section 3.2.) Following that, SSExploreAddNext() sets the depth of *nextState* to one plus the depth of *currentState* (Figure 3.2, line 11).

SSExploreAddNext() then checks whether *nextState* violates an assertion (Figure 3.2, line 12). We here distinguish between two disjoint types of events:

1. *safe event*: generates a *nextState* that does not violate any assertion.

```
 1. procedure SSExploreAddNext()
 2.    initialState.depth = 0 ;
 3.    AlreadyVisited.add(initialState) ;
 4.    ToBeExplored.add(initialState) ;
 5.    while ( ToBeExplored is not empty ) {
 6.       currentState = ToBeExplored.remove() ;
 7.       EnabledEvents = GenerateEnabledEvents(currentState) ;
 8.       for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
 9.          EventInfo E = EnabledEvents.get(i) ;
10.          nextState = GenerateNextState(currentState, E) ;
11.          nextState.setDepth(currentState.depth + 1) ;  /* set the depth of nextState */
12.          checkProperty = nextState.verifyAssertion() ;
13.          if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
14.             Print("Counterexample ") ;
15.             printCounterexample(nextState) ;
16.             exit ;
17.          } else if ( (checkProperty == true) AND (nextState.depth < MAXDEPTH) ) {
18.                if (nextState not simulated by any state in AlreadyVisited) { // use protocol-specific properties
19.                   if ( search strategy is BeFS ) {
20.                      nextState.computeBeFSTuple() ;  /* use protocol-specific properties */
                         }
21.                   AlreadyVisited.add(nextState) ;
22.                   ToBeExplored.add(nextState) ;
                   } /* end if */
                } /* end else if */
          }  /* end for */
       }  /* end while */

23. EventInfoList GenerateEnabledEvents(GlobalState currentState)
24.    EnabledEvents = { } ;
25.    for ( all protocol entities p ) { /* for all protocol entities */
26.       for ( all possible events e ) { /* for all events */
27.          NumberOfNextStates = EnablingFunction(currentState, p, e) ;
28.          for ( int i = 0 ; i < NumberOfNextStates ; i++ ) { /* for all integer-valued parameters */
29.             EnabledEvents.add(new EventInfo(p, e, i)) ;
             }  /* end for */
          }  /* end for */
       }  /* end for */
30.    return EnabledEvents ;

31. GlobalState GenerateNextState(GlobalState currentState, EventInfo E)
32.    CopyFromVModelToSModel(currentState) ;
33.    nextState = currentState ;  /* Start with nextState as a copy of currentState */
34.    ExecuteEvent(E) ;  /* Invoke E's event handler */
35.    CopyFromSModelToVModel(nextState) ;
36.    return nextState ;
```

Figure 3.2: An explicit-state stateful state space exploration procedure. This procedure adds a state being generated (i.e., *nextState*) to both *AlreadyVisited* (line 21) and *ToBeExplored* (line 22).

2. *unsafe event*: generates a *nextState* that violates an assertion.

Our state space exploration framework in J-Sim also allows the user to specify that a counterexample has to contain at least one state that is generated due to a particular event. This requirement is checked by calling the *DoesCounterexampleContainEvent()* function (Figure 3.2, line 13). (We have made use of this feature in some of our experiments in Chapters 4-6 and 8.) If the user does not want to make use of this feature, *DoesCounterexampleContainEvent()* always returns true.

If *nextState* violates an assertion (i.e., the case of an unsafe event) and *DoesCounterexample-ContainEvent()* returns true, a counterexample is printed by calling the *printCounterexample()* function (Figure 3.2, line 15), which is a recursive function that traces the state space backwards from *nextState* until *initialState* is reached.

If *nextState* does not violate an assertion (i.e., the case of a safe event) and the depth of *nextState* in the state space is strictly less than the specified maximum depth $MAXDEPTH$ (Figure 3.2, line 17), SSExploreAddNext() checks whether *nextState* is not simulated by any state in *AlreadyVisited* (Figure 3.2, line 18). We here distinguish between three disjoint types of events:

1. *deepest event*: generates a *nextState* whose depth is equal to $MAXDEPTH$.

2. *non-tree event*: generates a *nextState* whose depth is strictly less than $MAXDEPTH$ and that is simulated by a state s in *AlreadyVisited* such that the depth of s is less than or equal to the depth of *nextState*.

3. *tree event*: generates a *nextState* whose depth is strictly less than $MAXDEPTH$ and that is not simulated by any state s in *AlreadyVisited* such that the depth of s is less than or equal to the depth of *nextState*.

Note that it is impossible that the depth of *nextState* is strictly greater than $MAXDEPTH$. In Section 3.3.5, we provide an example that illustrates the definitions of tree, non-tree and deepest events.

If the event that generated *nextState* is a safe tree event, *nextState* is added to *ToBeExplored* (Figure 3.2, line 22) in order to be explored later and is also added to *AlreadyVisited* (Figure 3.2, line 21) to denote that it has been already visited. Adding a state to *ToBeExplored* or *AlreadyVisited* requires a function that creates a copy of a state (e.g., *clone*()).

3.3.3 Different Search Strategies

Depending on the order in which states are added to, and removed from, *ToBeExplored*, SSExplore-AddNext() can employ breadth-first (BFS), depth-first (DFS) and best-first (BeFS) search strategies. Precisely, in BFS, *ToBeExplored* is implemented as a first-in first-out (FIFO) queue; in DFS, *To-BeExplored* is implemented as a last-in first-out (LIFO) stack; whereas in BeFS, *ToBeExplored* is implemented as a priority queue. We call these three search strategies: BFS-AN, DFS-AN and BeFS-AN respectively[1].

A best-first search strategy is implemented by *state ranking*. Specifically, as part of the verification model, the user writes a function that maps each state to a BeFS tuple $< b_1, b_2, \ldots, b_{B-1}, b_B >$ (Figure 3.2, line 20) based on protocol-specific properties. The state space explorer then considers *ToBeExplored* as a priority queue in which states are ranked in decreasing lexicographical order of

[1] AN stands for "Add Next".

this tuple; i.e., a state s_1 is considered "better than" a state s_2 if s_1 has a higher lexicographical order of this tuple than s_2.

Clearly, both DFS and BeFS are not level-order traversals of the state space. Hence, in order to implement the stateful search correctly in these two cases, we limit the search in *AlreadyVisited* (Figure 3.2, line 18) to the states whose depth is less than or equal to the depth of *nextState*.

Note that SSExploreAddNext() adds a state being generated (i.e., *nextState*) to *AlreadyVisited* (Figure 3.2, line 21) only if *nextState* is not simulated by any state in *AlreadyVisited*. Another way of implementing the explicit-state stateful search is to add a state being explored (i.e., *currentState*) to *AlreadyVisited* only if *currentState* is not simulated by any state in *AlreadyVisited*. This stateful search, which we call SSExploreAddCurrent(), is shown in Figure 3.3. The difference between SSExploreAddNext() and SSExploreAddCurrent() is that the former eagerly detects and eliminates similar states while the latter lazily does so. In other words, SSExploreAddNext() emphasizes saving memory at the expense of spending time for checking whether *nextState* is simulated by any state in *AlreadyVisited* (Figure 3.2, line 18) whereas SSExploreAddCurrent() emphasizes saving the time of this check until is needed (Figure 3.3, line 7) at the expense of allocating memory for (possibly unnecessarily) adding *nextState* to *ToBeExplored* (Figure 3.3, line 22). The choice of eager versus lazy detection of similar states may hence affect the time and memory costs of the state space search. Since the number of similar states in a state space is generally unknown in advance and the time and memory budgets may differ from one case study to another, we provide both types of stateful searches in our framework. Similar to SSExploreAddCurrent(), SSExploreAddNext() can also employ BFS, DFS and BeFS search strategies depending on the order in which states are added to, and removed from, *ToBeExplored*. We call these three search strategies: BFS-AC, DFS-AC and BeFS-AC respectively[2].

A third way of implementing the explicit-state stateful search is a recursive depth-first search, which does not make use of *ToBeExplored* but instead uses the program's stack. We call this search strategy: DFS-R (shown in Figure 3.4)[3].

3.3.4 Different Randomized Search Strategies

The performance of each of the seven search strategies mentioned above depends on the order in which enabled events are added to the list of enabled events *EnabledEvents* (Figure 3.2, line 29). SSExploreAddCurrent(), SSExploreAddNext() and SSExploreRecursiveDFS() assume a fixed search

[2]AC stands for "Add Current".
[3]R stands for "Recursive".

```
 1. procedure SSExploreAddCurrent()
 2.    initialState.depth = 0 ;
 3.    AlreadyVisited = { } ;
 4.    ToBeExplored.add(initialState) ;
 5.    while ( ToBeExplored is not empty ) {
 6.       currentState = ToBeExplored.remove() ;
 7.       if (currentState not simulated by any state in AlreadyVisited) { // use protocol-specific properties
 8.          AlreadyVisited.add(currentState) ;
 9.          EnabledEvents = GenerateEnabledEvents(currentState) ;  /* See Figure 3.2 for GenerateEnabledEvents() */
10.          for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
11.             EventInfo E = EnabledEvents.get(i) ;
12.             nextState = GenerateNextState(currentState, E) ;  /* See Figure 3.2 for GenerateNextState() */
13.             nextState.setDepth(currentState.depth + 1) ;  /* set the depth of nextState */
14.             checkProperty = nextState.verifyAssertion() ;
15.             if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
16.                Print("Counterexample ") ;
17.                printCounterexample(nextState) ;
18.                exit ;
19.             } else if ( (checkProperty == true) AND (nextState.depth < MAXDEPTH) ) {
20.                   if ( search strategy is BeFS ) {
21.                      nextState.computeBeFSTuple() ;  /* use protocol-specific properties */
                      }
22.                   ToBeExplored.add(nextState) ;
                   } /* end else if */
             } /* end for */
          } /* end if */
       } /* end while */
```

Figure 3.3: An explicit-state stateful state space exploration procedure. This procedure adds a state being explored (i.e., *currentState*) to *AlreadyVisited* (line 8) and a state being generated (i.e., *nextState*) to *ToBeExplored* (line 22).

order; i.e., the order is the same each time the procedure executes. Specifically, the search order determined by the three for loops (Figure 3.2, lines 25-29) is: increasing order of protocol entity IDs p, increasing order of event IDs e and increasing order of event parameters i.

However, it has been shown in [30] that variations in the search order can give rise to very large variations in state space exploration costs and assertion violation detection effectiveness. In order to allow for search order variations, we implement randomized versions of SSExploreAddCurrent(), SSExploreAddNext() and SSExploreRecursiveDFS(). Similar to [29], randomization is achieved by shuffling the set of enabled events at each state being explored using a Fisher-Yates shuffling algorithm [56]. Hence, the order of enabled events in *EnabledEvents* is randomized each time the function *GenerateEnabledEvents()* executes (Figure 3.2 (line 7), Figure 3.3 (line 9) and Figure 3.4 (line 8)). Randomization in the shuffle follows a pseudo-random sequence whose seed is passed as a parameter to the state space exploration framework. We call the corresponding seven randomized search strategies: BFS-ANS, DFS-ANS, BeFS-ANS, BFS-ACS, DFS-ACS, BeFS-ACS and DFS-RS.[4] We do not compare between the randomized and non-randomized versions of the search strategies in this thesis.

[4]ANS, ACS and RS respectively stand for "Add Next with Shuffle", "Add Current with Shuffle" and "Recursive with Shuffle".

```
 1. procedure SSExploreRecursiveDFS()
 2.    initialState.depth = 0 ;
 3.    AlreadyVisited.add(initialState) ;
 4.    RecursiveDFS(initialState) ;

 5. procedure RecursiveDFS(GlobalState s)
 6.    currentState = s ;
 7.    temp = currentState ;  /* save a copy of current state */
 8.    EnabledEvents = GenerateEnabledEvents(currentState) ;  /* See Figure 3.2 for GenerateEnabledEvents() */
 9.    for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
10.       EventInfo E = EnabledEvents.get(i) ;
11.       nextState = GenerateNextState(currentState, E) ;  /* See Figure 3.2 for GenerateNextState() */
12.       nextState.setDepth(currentState.depth + 1) ;  /* set the depth of nextState */
13.       checkProperty = nextState.verifyAssertion() ;
14.       if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
15.          Print("Counterexample ") ;
16.          printCounterexample(nextState) ;
17.          exit ;
18.       } else if ( (checkProperty == true) AND (nextState.depth < MAXDEPTH) ) {
19.            if (nextState not simulated by any state in AlreadyVisited) { // use protocol-specific properties
20.               AlreadyVisited.add(nextState) ;
21.               RecursiveDFS(nextState) ;
22.               currentState = temp ;  /* restore current state */
               } /* end if */
            } /* end else if */
      } /* end for */
```

Figure 3.4: An explicit-state stateful state space exploration procedure. SSExploreRecursiveDFS() employs a recursive depth-first search that does not make use of *ToBeExplored*. This procedure adds a state being generated (i.e., *nextState*) to *AlreadyVisited* (line 20).

3.3.5 An Example

In this section, we provide an example of a state space graph and some examples of its exploration. Consider the state space graph shown in Figure 3.5. s_0 is the initial state. Each state is simulated by itself since a simulation relation is reflexive. Assume further that s_4 is simulated by s_2 and s_7 is simulated by s_5. For simplicity, we assume that none of the states violates an assertion.

Figure 3.6 shows four different state space explorations of the state space graph in Figure 3.5. We assume that $MAXDEPTH = 3$. The order in which a state is visited is shown at the top left corner of a state. Solid edges correspond to tree events, dashed edges correspond to non-tree events, and dotted edges correspond to deepest events. The stateful search turned the state space graph into a tree. The initial state is the root of the tree. The states generated by the tree events are the interior nodes while the states generated by the non-tree and deepest events are the leaves of the tree. Non-tree events make the search stateful. Deepest and non-tree events make the tree finite.

3.4 Implementation Problems and Solutions

We have encountered two major implementation problems in the course of incorporating the state space explorer into J-Sim: one is related to how network protocol entities communicate with each other, with the state space explorer in between; and the other is related to the ACA timers. In this

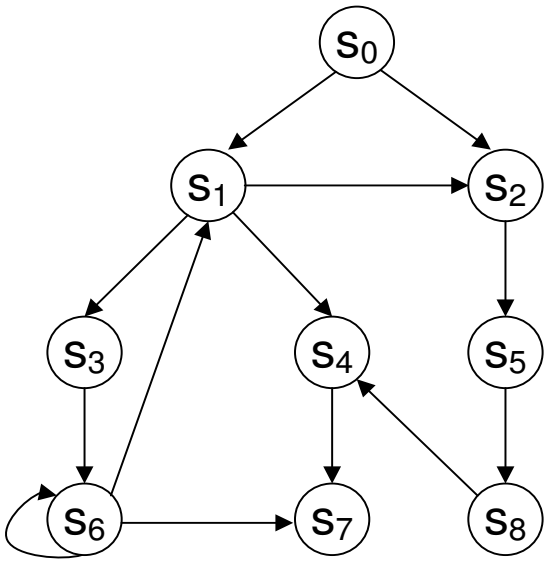
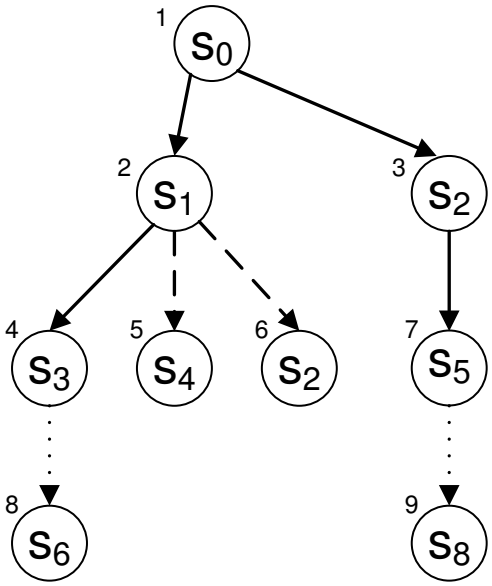


Figure 3.5: An example state space graph. s_0 is the initial state. Each state is simulated by itself since a simulation relation is reflexive. Assume further that s_4 is simulated by s_2 and s_7 is simulated by s_5.

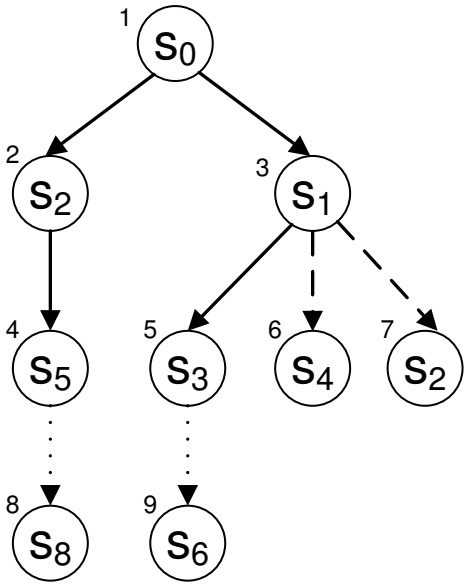
section, we describe both problems and how we solve them while keeping our two design goals met.

Without verification using state space exploration, the protocol entities communicate with each other via ports. However, when the simulation model is being verified and the state space explorer is used as shown in Figure 3.1, the protocol entities need to communicate with each other via the state space explorer. Initially, we simply connected the ports of each protocol entity to those of the state space explorer, but then found that protocol-specific data/control messages generated by the protocol entities during the execution of an event handler (Figure 3.2, line 34) may not be forwarded to the state space explorer at the required time. This is because the state space explorer does not wait until the protocol entities finish executing an event handler. This may cause the state space explorer to exclude some of the new state's information in *nextState* (Figure 3.2, line 35). We solve this problem by setting the ports that are involved in the interaction between the state space explorer and the protocol entities to the *function-call execution model* instead of the default *independence execution model* [108]. Figure 3.7 shows the two execution models supported by the ACA. In the function-call execution model, the state space explorer (Component A in Figure 3.7) *waits* until a protocol entity (Component B in Figure 3.7) finishes executing an event handler; therefore, this solution ensures that all the new state's information will be included in *nextState*.

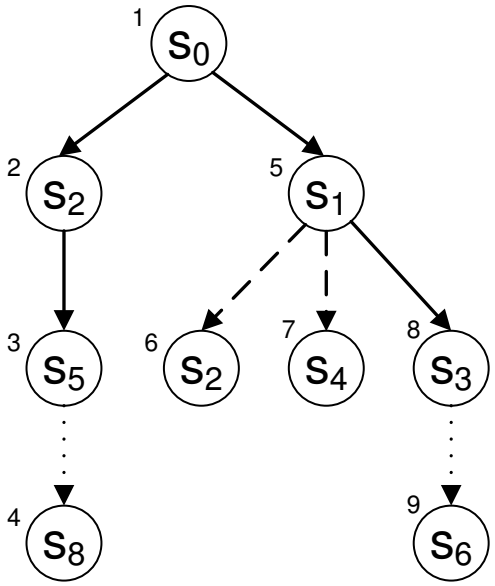
However, this solution requires modifying the J-Sim simulation model of the network protocol;



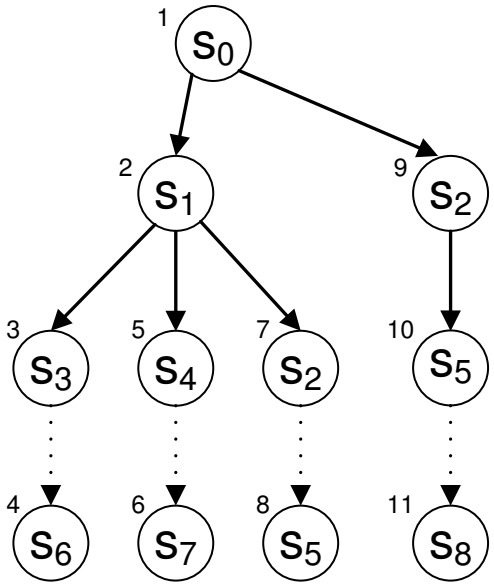
(a) A BFS-ANS exploration.



(b) Another BFS-ANS exploration.

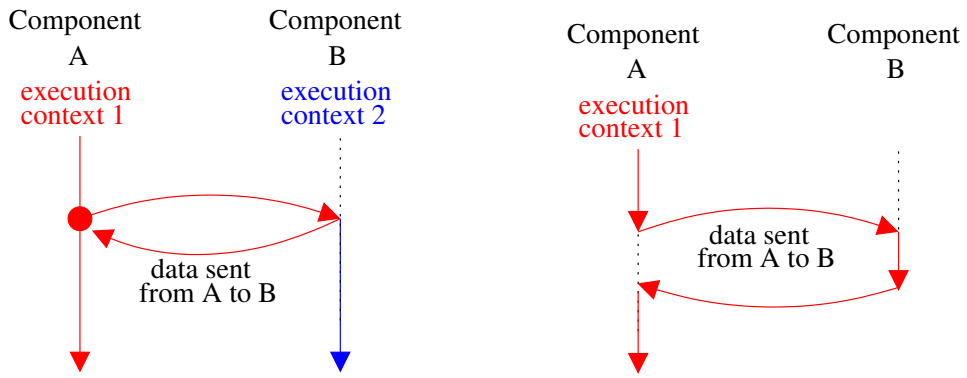


(c) A DFS-RS exploration.



(d) Another DFS-RS exploration.

Figure 3.6: Examples of state space explorations of the state space graph shown in Figure 3.5. $MAXDEPTH = 3$. The order in which a state is visited is shown at the top left corner of a state. Solid edges correspond to tree events, dashed edges correspond to non-tree events, and dotted edges correspond to deepest events. The depth of a state is given by its level in a figure.



(a) Independence execution model

(b) Function-call execution model

Figure 3.7: The execution models supported by the ACA. (This figure is excerpted from [108].)

hence, it might violate our second design goal. In order to keep our design goals met, we prefer making this modification in a subclass of the Java class that implements the simulation model of a network protocol entity, thus keeping the original parent Java class unmodified and ensuring that the J-Sim simulation model execution time will not degrade if the user is only interested in using the model for performance evaluation purposes. For a similar reason, we set the ports that are involved in the interaction between the state space explorer and the global states to the *function-call execution model*.

The second problem is related to the ACA timers, which are used to model timers (e.g., retransmission timers) in network simulation models in J-Sim. Without verification using state space exploration, a protocol entity that uses an ACA timer sets the timer to a pre-determined time interval. When the timer expires, a `timeout()` callback function is invoked to handle the timeout event if the timer is still active (i.e., has not already been canceled). If the simulation model is to be verified, the state space explorer should explore all the possible events from a given state, and should not be limited to a single timeout value for each timer. Instead, the state space explorer should trigger the timeout event when that event may occur in the real world. For example, a typical retransmission timer in a reliable unicast protocol may expire at any time as long as there is a pending data message that has been sent but not yet acknowledged. (We assume that the setting of the interval of a timer may differ from one run of the simulation to another (which is typically the case, especially if a pseudo-random number sequence is used for generating the interval of a timer); otherwise, this approach may suffer from excessive false positives.)

A possible solution to this problem is to modify the implementation of the *setTimeout()* method defined in class *Module*, a key J-Sim class that has a *timer port* used to set up (and cancel) timers. However, this solution requires modifying the core implementation of J-Sim, and hence violates our first design goal. In order to keep our design goals met, we choose to make this modification in a subclass of the Java class that implements the simulation model of a network protocol entity.

3.5 Role of the User

It should be noticed that the state space exploration process is not fully automated. We summarize below what the user needs to do, in order to verify the simulation model of a network protocol.

1. **States:** Provide an implementation of *GlobalState* (including writing the assertion as a Java method and a function that creates a copy of a state), and specify how to construct the initial state. To reduce the user's burden, we provide an implementation of a class, called *SystemState*, that includes the protocol-independent information (e.g., the depth of a state and which event generated the state). *GlobalState*, which should be implemented as a sub-class of *SystemState*, includes the protocol-specific information.

2. **Events:** Specify (a) the set of events that exist in the network protocol, (b) the *Enabling-Function()* that returns the state-dependent number of possible successor states that an event generates; zero if the event is disabled, and (c) how each event is handled. (As mentioned above, the user has to write the event handlers anyway in order to have a working simulation model of the network protocol in J-Sim.)

3. **Enabling the Stateful Search:** In order to enable the stateful search, the user needs to implement either $isSimulatedBy()$ or $computeHashCode()$; the former is needed only if *AlreadyVisited* stores the states that have already been visited whereas the latter is needed only if *AlreadyVisited* stores the hash codes of the states that have already been visited. $isSimulatedBy()$ determines based on protocol-specific properties, whether two states are similar. Observe that there is a general simulation relation that can be exploited when the communication is assumed to be unreliable (Section 3.3.1). $computeHashCode()$ computes the hash code of a state.

4. **State Ranking:** In order to enable a best-first search strategy, the user needs to write the Java method $computeBeFSTuple()$ (Figure 3.2 (line 20) and Figure 3.3 (line 21)) that maps

each state to a tuple $< b_1, b_2, \ldots, b_{B-1}, b_B >$ based on protocol-specific properties such that a state s_1 is considered "better than" a state s_2 if s_1 has a higher lexicographical order of this tuple than s_2.

5. **Interaction with the State Space Explorer:** Provide implementations for the operations that involve information exchange with the state space explorer component (e.g., *CopyFromVModelToSModel()* and *CopyFromSModelToVModel()*). To facilitate programming, we made use of ports and contracts to provide a seamless interface between components; in this case, between the state space explorer on one side and either the protocol entities or the global states on the other side (see Figure 3.1). In addition, the user needs to modify the Java class that implements the simulation model of a network protocol entity (or preferably its subclass), as explained in Section 3.4, to facilitate interaction with the state space explorer.

It is important to note that making use of protocol-specific properties for the simulation relation and state ranking is done in the verification model and is hence isolated from both the state space explorer and the simulation model. This ensures that the state space exploration framework is general enough and not tied to a particular network protocol or communication mechanism. Furthermore, this allows the user to try several simulation relations and/or state ranking mechanisms until a counterexample is found within a certain budget of time and/or memory. In the following chapters, we elaborate further on these points.

Chapter 4

AODV Case Study

In this chapter, we apply the J-Sim state space exploration framework to the J-Sim simulation model of the Ad-hoc On-Demand Distance Vector routing protocol for wireless ad-hoc networks. In Section 4.1, we give an overview of AODV's key functionality. In Section 4.2, we describe the steps that we follow to verify its simulation model including how we exploit protocol-specific properties. In Section 4.3, we present the results of this verification. In Section 4.4, we compare the performance of our state space exploration framework in J-Sim to that of Java PathFinder (JPF) [52, 111], a state-of-the-art model checker for Java programs. Finally, in Section 4.5, we summarize the lessons that we learned in this case study.

The J-Sim simulation model of AODV (not including the J-Sim library) has about 1200 lines of code. We conduct all the experiments on a dual-processor Intel Xeon 2.8 GHz machine running Linux version 2.6.17 with 2 GB memory. We use Sun's 1.5.0 04-b05 Java HotSpot^{TM} Client VM with 0.5 GB initial heap size and 1.5 GB maximum heap size.

4.1 Overview of AODV

The Ad-hoc On-Demand Distance Vector (AODV) routing protocol [78] is a well-known and widely used reactive routing protocol for multihop wireless ad-hoc networks. AODV is reactive in the sense that a route to a given destination is established via a route discovery process only when it is needed by a source node (i.e., traffic-driven). In this section, we describe the J-Sim simulation model of AODV, which is based on AODV Draft (version 11) [77].

In AODV, each node n in the ad-hoc network maintains a routing table. For node n, a routing table entry (RTE) to a destination node d contains the following fields: a destination address $dest_{n,d}$ (the address of the destination d), a next hop address $nexthop_{n,d}$ (the address of the node to which n forwards data packets destined for node d), a hop count $hops_{n,d}$ (the number of hops needed to reach node d from node n), a destination sequence number $seqno_{n,d}$ (a measure of the freshness of

the route information), and a flag $isValid_{n,d}$ (representing whether the RTE is valid or invalid).

Each RTE is associated with a lifetime. Periodically, a route timeout event is triggered invalidating (but not deleting) all the RTEs that have not been used (e.g., to send or forward packets to the destination) for a time interval that is greater than the lifetime. Invalidating a RTE involves setting $isValid_{n,d}$ to false, incrementing $seqno_{n,d}$ and setting $hops_{n,d}$ to ∞.

Each node n also maintains two monotonically increasing counters: a node sequence number $seqno_n$ (whose initial value is 2) and a broadcast ID bid_n (whose initial value is 1). When node n requires a route to a destination d, if it does not already have a valid RTE to node d, it first creates an invalid RTE to node d with $hops_{n,d}$ set to ∞ and $seqno_{n,d}$ set to zero. Following that, node n increments $seqno_n$, *broadcasts* a route request (RREQ) packet containing the following fields $< n, seqno_n, bid_n, d, seqno_{n,d}, hopCount_q >$ and then increments bid_n. The $hopCount_q$ field is initialized to 1. The pair $< n, bid_n >$ uniquely identifies a RREQ packet.

Each node m, receiving the RREQ packet from node n, keeps the pair $< n, bid_n >$ in a broadcast ID cache so that it can later check if it has already received a RREQ with the same source address and broadcast ID. If so, the incoming RREQ packet is discarded. If not, node m either satisfies the RREQ by *unicasting* a route reply (RREP) packet back to node n if it has a fresh enough route to node d (or it is node d itself), or rebroadcasts the RREQ to its own neighbors after incrementing the $hopCount_q$ field if it does not have a fresh enough route to node d (nor is itself node d). An intermediate node m determines whether it has a fresh enough route to node d by comparing the destination sequence number $seqno_{m,d}$ in its own RTE with the $seqno_{n,d}$ field in the RREQ packet.

Each intermediate node also records a reverse route to the requesting node n; this reverse route will be used to send/forward route replies to node n. The requesting node's sequence number $seqno_n$ is used to maintain the freshness of this reverse route. Each entry in the broadcast ID cache has a lifetime. Periodically, a broadcast ID timeout event is triggered causing the deletion of entries in the cache that have expired.

A RREP packet, which is sent by an intermediate node m, contains the following fields $< n, d, seqno_{m,d}, hopCount_p >$. The $hopCount_p$ field is initialized to $1+hops_{m,d}$. If it is the destination d that sends the RREP packet, it first increments $seqno_d$ and then sends a RREP packet containing the following fields $< n, d, seqno_d, 1 >$. The unicast RREP travels back to the requesting node n via the reverse route. Each intermediate node along the reverse route sets up a forward pointer to the node from which the RREP came, thus establishing a forward route to the destination d, increments the $hopCount_p$ field and forwards the RREP packet to the next hop towards n.

If node m offers node n a new route to node d, node n compares $seqno_{m,d}$ (the destination sequence number of the offered route) to $seqno_{n,d}$ (the destination sequence number of the current route), and accepts the route with the greater sequence number. If the sequence numbers are equal, the offered route is accepted only if it has a smaller hop count than the hop count in the RTE; i.e., $hops_{n,d} > hops_{m,d}$.

4.2 Verifying the Simulation Model of AODV

We next present the steps that we follow to verify the J-Sim simulation model of AODV. These steps constitute a generic methodology for verifying the simulation model of any network protocol in J-Sim.

Step 1. States: Definitions of the global state, the initial state, and the assertion

We define *GlobalState* as a tuple that has two components, namely the protocol state and the network cloud. The protocol state is an array of N node states where N is the number of nodes in the ad-hoc network. The node state of a node n includes n's routing table, broadcast ID cache, $seqno_n$ and bid_n. The network cloud models the network as an unbounded multiset that contains AODV packets, and also maintains the neighborhood information. A broadcast AODV packet whose source is node s is modeled as a set of packets, each of which is destined for one of the neighbors (i.e., the nodes that are within the transmission range) of node s.

In the initial global state, the network does not contain any packets and the AODV process at each node is initialized as specified by the J-Sim simulation model of AODV. Specifically, the AODV process starts with an empty routing table, empty broadcast ID cache, $seqno_n = 2$ and $bid_n = 1$.

An important assertion in a routing protocol such as AODV is the *loop-free* property. Intuitively, a node must not occur at two points on a path between two other nodes; therefore, at each hop along a path from a node n to a destination d, either the destination sequence number must increase or the hop count toward the destination must decrease. Formally, consider two nodes n and m such that both nodes have valid RTEs to some destination d, and m is the next hop of n to d; i.e., $nexthop_{n,d} = m$. The loop-free property can be expressed as follows [16, 68]:

$$((seqno_{n,d} < seqno_{m,d}) \vee (seqno_{n,d} == seqno_{m,d} \wedge hops_{n,d} > hops_{m,d}))$$

Informally, the loop-free property requires that either m has a more fresh RTE to d than that of n (i.e., an RTE with a greater destination sequence number) or m has a shorter route that is equally fresh.

Step 2. Events

Next, we specify the set of events, when each event is enabled and the corresponding *EnablingFunction()*, and how each event is handled. We classify the events into two categories: node events (i.e., events that are triggered inside a node) and network events (i.e., events that are triggered inside the network). The events in each category are listed as follows:

1. Node Events

   T_0 Initiation of a route request by node n to a destination $d \neq n$: This event is enabled if node n does not have a valid RTE to the destination d. When enabled, *EnablingFunction(currentState, n, T_0)* returns 1. The event is handled by creating an invalid RTE to node d with $hops_{n,d}$ set to ∞, incrementing $seqno_n$, and then broadcasting a RREQ.

   T_1 Restart of the AODV process at node n: This event may take place because of a node reboot. This event is always enabled; i.e., *EnablingFunction(currentState, n, T_1)* always returns 1. The event is handled by reinitializing the state of the AODV process at node n.

   T_2 Broadcast ID timeout at node n: This event is enabled if there is at least one entry in the broadcast ID cache of node n. When enabled, *EnablingFunction(currentState, n, T_2)* returns the number of entries in the broadcast ID cache of node n. The event is handled by deleting an entry from the broadcast ID cache of node n.

   T_3 Timeout of the route to destination d at node $n \neq d$: This event is enabled if node n has a valid RTE to node d. When enabled, *EnablingFunction(currentState, n, T_3)* returns 1. The event is handled by invalidating this RTE.

2. Network Events

   T_4 Delivering an AODV packet to node n: This event is enabled if the network contains at least one AODV packet such that node n is the destination (or the next hop towards the destination) of the packet and node n is one of the neighbors of the source of the packet. When enabled, *EnablingFunction(currentState, n, T_4)* returns the number of the AODV

packets that satisfy these conditions. The event is handled by removing one of these AODV packets from the network and forwarding it to node n in order to be processed as explained in Section 4.1.

T_5 Loss of an AODV packet destined for node n: This event is enabled if the network contains at least one AODV packet that is destined for node n. When enabled, *Enabling-Function(currentState, n, T_5)* returns the number of the AODV packets that satisfy this condition. The event is handled by removing one of these AODV packets from the network.

Step 3. Simulation Relation

We use the general simulation relation outlined in the introduction and Section 3.3. For AODV specifically it reduces to the following definition. A state s_2 is said to simulate a state s_1 if (i) s_1 and s_2 have the same neighborhood information, (ii) for each AODV packet in s_1, there is a corresponding equivalent AODV packet in s_2, and (iii) for each node n, s_1 and s_2 have equal corresponding values for $seqno_n$, bid_n, and node n's routing table and broadcast ID cache (each viewed as an unordered set of entries).

In Appendix A, we argue that this relation is a simulation relation.

Step 4. State Ranking: Exploiting protocol-specific properties

Recall from Chapter 3 that in order to enable a best-first search strategy, the user needs to write a Java method that maps each state to a tuple $< b_1, b_2, \ldots, b_{B-1}, b_B >$ based on protocol-specific properties such that a state s_1 is considered "better than" a state s_2 if s_1 has a higher lexicographical order of this tuple than s_2.

A suitable BeFS heuristic for exploring the state space of AODV can be obtained by inspecting the loop-free property. A node, which does not have a valid RTE to any node, does not affect the truth value of the loop-free property. Therefore, a suitable BeFS heuristic (which we call AODV-1-BeFS) is to consider a state s_1 better than a state s_2 if the number of *valid* RTEs to any node in s_1 is greater than that in s_2. In other words, the BeFS tuple, assigned to a state s, consists of one component $< b_1 >$ such that b_1 is the number of valid RTEs to any node in s.

Another BeFS heuristic (which we call AODV-2-BeFS) is obtained by inspecting the loop-free property, which can be rewritten as follows:

$$(((seqno_{n,d} - seqno_{m,d}) < 0) \vee (seqno_{n,d} == seqno_{m,d} \wedge ((hops_{m,d} - hops_{n,d}) < 0)))$$

Therefore, the greater $(seqno_{n,d} - seqno_{m,d})$ and/or $(hops_{m,d} - hops_{n,d})$ in a state s, the more likely s is close to an assertion violation. Hence, AODV-2-BeFS considers a state s_1 better than a state s_2 if the following summation

$$M_d = \sum_{n \neq d}((seqno_{n,d} - seqno_{m,d}) + (hops_{m,d} - hops_{n,d}))$$

in s_1 is greater than that in s_2, where $nexthop_{n,d} = m$. In other words, the BeFS tuple, assigned to a state s, consists of one component $< b_1 >$ such that $b_1 = M_d$. Note that the summation M_d is specific to a certain destination d; hence, AODV-2-BeFS requires considering only one destination node (as we do in Section 4.3). The summation M_d includes only the nodes n and m that have valid RTEs to the destination d. If none of the nodes have a valid RTE to node d, M_d is set to $-\infty$ (i.e., worst or least interesting state).

In addition to AODV-1-BeFS and AODV-2-BeFS, we also consider the following BeFS heuristics:

1. AODV-3-BeFS: Similar to AODV-2-BeFS, AODV-3-BeFS requires considering only one destination node d. This heuristic assigns to a state s a BeFS tuple that consists of two components $< b_1, b_2 >$ such that b_1 is the number of valid RTEs *to the destination* d in s, and b_2 is the number of valid RTEs *to any node* in s.

2. AODV-4-BeFS: Since a valid RTE is established upon receiving a RREP packet, AODV-4-BeFS assigns to a state s a BeFS tuple that consists of one component $< b_1 >$ such that b_1 is the number of RREP packets in s.

3. AODV-5-BeFS: AODV-5-BeFS assigns to a state s a BeFS tuple that consists of two components $< b_1, b_2 >$ such that b_1 is the number of RREP packets in s (as in AODV-4-BeFS), and b_2 is the number of valid RTEs to any node in s.

4.3 Results of the Verification

Clearly, the state space created by the J-Sim simulation model of AODV is infinite. Furthermore, there is an infinite number of possible initial states depending on the number of nodes and the network topology. As an attempt towards handling the state space explosion problem, we (1) consider an initial state of an ad-hoc network consisting of N nodes: n_0, n_1, ..., n_{N-1} arranged in a chain topology where each node is a neighbor of both the node to its left and the node to its right (if any exists); i.e., all wireless links are assumed to be bidirectional, and (2) reduce the number of events and states by considering only one destination node n_{N-1}. Therefore, all RREQ

packets request a route to node n_{N-1} and the route timeout event invalidates the RTE to node n_{N-1} only. Furthermore, the loop-free property checks the absence of routing loops to node n_{N-1} only. Although this scenario is simple, it ensures that nodes $n_0, n_1, \ldots, n_{N-3}$ require multihop routes to reach node n_{N-1}; i.e., AODV multihop routing is needed. In addition, if an assertion is violated in a chain network topology, it may also be violated in an arbitrary network topology.

While verifying the J-Sim simulation model of AODV in a chain network topology consisting of $N = 3$ nodes, we have discovered an assertion violation (which we call Counterexample 1) in the AODV simulation model caused by its failure to follow a part of the AODV specification [77] that determines certain actions that must be taken after a node reboot. Conceptually, if $nexthop_{0,2} = 1$ and the AODV process at node n_1 restarts due to a node reboot, the net effect is that all the RTEs stored at node n_1 will be deleted. As a result, node n_1 may later accept a route that was offered by node n_2 with a lower sequence number than that of node n_0 (i.e., $seqno_{0,2} > seqno_{1,2}$), hence violating the loop-free property. We also manually injected two errors (which we call Counterexamples 2 and 3 respectively): in Counterexample 2, $seqno_{n,d}$ is not incremented when a RTE is invalidated and in Counterexample 3, a RTE is deleted (instead of invalidated) when its lifetime expires. The state space exploration framework was able to find these two errors too.[1] A routing loop may occur due to either of these two errors. This is because in the case that $nexthop_{0,2} = 1$ and a route timeout event takes place at node n_1, in either Counterexample 2 or 3, if n_1 is later offered a route to node n_2 by node n_0, this route will be accepted (because in Counterexample 2, $hops_{1,2} = \infty$; hence, $hops_{1,2} > hops_{0,2}$; whereas in Counterexample 3, $seqno_{0,2} > seqno_{1,2}$). The interested reader is referred to Appendix B for detailed traces (along with the explanations) of the three counterexamples.

Table 4.1 gives the performance of the various randomized search strategies, for finding each of the three counterexamples, with respect to the following two types of performance evaluation criteria: (a) *platform-independent*; namely, the number of events executed and the number of states stored in memory (sum of the sizes of *AlreadyVisited* and *ToBeExplored*)[2], and (b) *platform-dependent*; namely, the time needed to find an assertion violation. In Table 4.1, *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation. We ran 100 experiments for each search strategy. Each experiment has a different seed, but the same set

[1]For Counterexamples 2 and 3, we require that the counterexample contain at least one state that is generated due to the route timeout event, T_3. In order to achieve that, we made use of the *DoesCounterexampleContainEvent()* function provided by the state space exploration framework (Figures 3.2-3.4).

[2]Note that we do not report the total number of states generated because it is simply equal to the number of events executed plus 1 (to account for the initial state).

of 100 seeds were used for each of the seven search strategies. For each performance evaluation criterion, we report the minimum, the maximum and the average values.

Counterexample 1	Time (s.)			Space (number of states)			Events		
$MAXDEPTH = 10$	Min	Max	Average	Min	Max	Average	Min	Max	Average
DFS-RS	0.290	14.773	4.727	117	2069	971.09	1175	30669	13566.29
BFS-ACS	7.445	25.013	15.854	10112	19463	15206.02	14513	28182	21933.33
DFS-ACS	0.133	15.577	4.751	98	2130	1018.12	285	30530	13542.06
AODV-1-BeFS-ACS	0.477	4.142	1.246	423	1587	727.86	1900	13123	4898.39
AODV-2-BeFS-ACS	0.192	16.592	4.165	157	2176	908.16	411	31310	11102.91
AODV-3-BeFS-ACS	0.347	4.278	1.030	280	1594	609.81	1148	13433	3746.18
AODV-4-BeFS-ACS	0.375	6.769	2.322	262	2266	1042.60	1463	19004	9040.47
AODV-5-BeFS-ACS	0.489	3.041	1.611	319	1628	1034.66	2063	11562	6958.73
BFS-ANS	35.552	129.806	79.759	4475	8101	6430.43	14513	28182	21933.33
DFS-ANS	0.142	14.422	4.671	94	1961	979.60	251	27416	13268.05
AODV-1-BeFS-ANS	0.483	5.351	1.463	440	1542	726.72	1633	12432	4575.70
AODV-2-BeFS-ANS	0.183	16.219	3.899	176	2148	888.96	355	29793	10373.95
AODV-3-BeFS-ANS	0.343	5.379	1.202	345	1564	627.70	1091	12868	3529.72
AODV-4-BeFS-ANS	0.372	8.237	2.603	195	1978	913.52	1442	17638	8372.29
AODV-5-BeFS-ANS	0.493	4.328	2.207	236	1529	947.65	2008	11384	6778.21
Counterexample 2	Time (s.)			Space (number of states)			Events		
$MAXDEPTH = 10$	Min	Max	Average	Min	Max	Average	Min	Max	Average
DFS-RS	0.169	37.138	9.439	47	3492	1432.37	381	49553	20355.26
BFS-ACS	7.819	27.636	17.280	10056	20321	15928.23	14428	29704	23030.89
DFS-ACS	0.397	32.664	9.342	198	3339	1472.73	1661	46158	20222.31
AODV-1-BeFS-ACS	0.088	15.788	4.034	84	2448	866.31	97	30166	8997.88
AODV-2-BeFS-ACS	0.210	39.989	6.974	169	3647	1174.64	492	51171	14738.62
AODV-3-BeFS-ACS	0.098	15.726	3.557	97	2486	692.41	124	30295	7218.30
AODV-4-BeFS-ACS	0.807	11.692	6.313	688	2851	1855.61	3933	26449	17422.37
AODV-5-BeFS-ACS	0.426	8.512	2.819	345	2530	1158.91	1613	21288	8679.10
BFS-ANS	36.106	130.238	82.658	4484	8370	6712.08	14428	29704	23030.89
DFS-ANS	0.238	35.339	8.927	123	3342	1380.25	764	47306	19133.05
AODV-1-BeFS-ANS	0.120	17.817	4.530	97	2399	852.34	97	30088	8772.99
AODV-2-BeFS-ANS	0.209	35.715	6.881	174	3510	1175.56	480	48319	14076.26
AODV-3-BeFS-ANS	0.125	17.526	3.892	108	2425	695.41	112	29830	7059.34
AODV-4-BeFS-ANS	1.007	15.618	7.591	622	2638	1665.14	3747	26278	16729.05
AODV-5-BeFS-ANS	0.444	11.444	3.752	329	2248	1014.04	1563	21345	8516.39
Counterexample 3	Time (s.)			Space (number of states)			Events		
$MAXDEPTH = 10$	Min	Max	Average	Min	Max	Average	Min	Max	Average
DFS-RS	0.228	13.211	4.794	80	1899	967.88	796	28678	13916.49
BFS-ACS	7.395	23.643	16.295	10140	18560	15181.45	14662	27348	22231.78
DFS-ACS	0.155	17.913	4.435	94	2065	960.95	329	28543	12902.70
AODV-1-BeFS-ACS	0.089	3.397	1.080	84	1490	582.81	102	11995	4119.79
AODV-2-BeFS-ACS	0.215	14.788	3.353	149	2081	802.72	600	28588	9441.64
AODV-3-BeFS-ACS	0.096	3.531	0.837	95	1494	459.12	126	12366	2968.35
AODV-4-BeFS-ACS	0.380	5.405	1.911	257	1914	907.43	1480	15894	7844.89
AODV-5-BeFS-ACS	0.421	1.266	0.681	303	848	506.62	1617	6022	3043.93
BFS-ANS	32.710	107.542	73.664	4378	7426	6163.44	14662	27348	22231.78
DFS-ANS	0.154	16.334	3.973	90	1955	877.15	305	27291	11896.59
AODV-1-BeFS-ANS	0.119	4.138	1.290	97	1427	568.49	97	11583	3909.00
AODV-2-BeFS-ANS	0.209	11.673	3.085	151	1821	796.99	557	24905	8819.96
AODV-3-BeFS-ANS	0.125	4.318	0.972	108	1403	453.03	116	11915	2797.03
AODV-4-BeFS-ANS	0.372	7.219	2.159	209	1795	810.34	1467	16239	7457.34
AODV-5-BeFS-ANS	0.441	1.602	0.789	240	816	465.90	1571	5990	2993.95

Table 4.1: AODV case study: Time and space requirements (sum of the sizes of *AlreadyVisited* and *ToBeExplored*) and the number of events executed for finding the three counterexamples in a 3-node chain ad-hoc network using several randomized search strategies. *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation.

As shown in Table 4.1, AODV-1-BeFS-ACS (AODV-1-BeFS-ANS)[3] achieves significant reduction with respect to the evaluation criteria when compared to other standard search strategies such as BFS-ACS and DFS-ACS (BFS-ANS and DFS-ANS) respectively. Also, the choice of the BeFS

[3]We explain the notation as follows: AODV-1-BeFS denotes the BeFS heuristic itself whereas AODV-1-BeFS-ACS denotes the BeFS-ACS search strategy when making use of the AODV-1-BeFS heuristic.

heuristic has an impact on the performance. As shown in Table 4.1, AODV-2-BeFS-ACS (AODV-2-BeFS-ANS) performs worse than AODV-1-BeFS-ACS (AODV-1-BeFS-ANS) for the three counterexamples. This is because AODV-2-BeFS requires that a node (and its next hop towards the destination) have valid RTEs to the destination. This may not be true in the first few stages (i.e., lower depths) of the search space. Therefore, in the first few stages of the search, the non-visited states may look equally good and thus, AODV-2-BeFS may not be able to explore the states that are most likely to lead to the assertion violation first. AODV-3-BeFS tackles this problem by further differentiating equally good states by using a two-level BeFS heuristic. Hence, as shown in Table 4.1, AODV-3-BeFS-ACS and AODV-5-BeFS-ACS (AODV-3-BeFS-ANS and AODV-5-BeFS-ANS) respectively outperform AODV-2-BeFS-ACS and AODV-4-BeFS-ACS (AODV-2-BeFS-ANS and AODV-4-BeFS-ANS) because they are able to better guide the best-first search towards the assertion violation even at the lower depths of the search space.

4.4 Comparison with the Java PathFinder (JPF) Model Checker

To further evaluate the state space exploration framework in J-Sim, we compare its performance to that of Java PathFinder (JPF) [52, 111], a state-of-the-art model checker for Java programs. In this section, we first give an overview of JPF (version 4.1). Following that, we elaborate on how we enable JPF to explore the state space of the J-Sim simulation model of AODV. Finally, we present the performance results [90].

4.4.1 Overview of JPF

JPF is a general-purpose explicit-state model checker for Java programs. JPF takes as input a Java program and an optional bound on the length of program execution. JPF explores all executions (up to the given bound) that the program can have due to different thread interleavings and nondeterministic choices. JPF can generate as output the traces of those executions that violate a given property; e.g., the loop-free assertion in AODV. JPF is implemented in Java as a special Java Virtual Machine (JVM) that runs on top of the host JVM.[4] The main difference between JPF and a regular JVM is that JPF implements a *backtrackable* JVM; i.e., JPF can (quickly) backtrack the program execution by restoring a state that was previously encountered during the execution.

[4] In contrast to JPF, our state space exploration framework in J-Sim does not require a special JVM.

Backtracking allows exploration of different executions from the same state.

To achieve fast backtracking, JPF uses a *special representation of states* and executes program bytecodes by modifying this representation. For instance, JPF represents the heap as follows: JPF uses integers to represent object identifiers and to encode all field values, be they primitive (int, boolean, float, etc.) or pointers to other objects (which can hold the special value null). (JPF determines the meaning of various integers based on the field types kept in the class information.) Conceptually, JPF represents each object as an integer array, and the entire heap as an array of integer arrays. This special representation enables JPF to quickly store and restore states for backtracking; it is crucial for making the overall state space exploration efficient.

In order to enable a stateful search, JPF computes a linear representation of the state, also called *state linearization* [49], by traversing the root of the heap in a depth-first search order, assigning a unique identifier to each object (null pointers have value 0) and backtracking when it detects a cycle. Linearization builds a canonical representation of the state and hence enables the comparison of states during the state space exploration; two states are equivalent if their linearizations are equal. Specifically, JPF computes a hash code from the state linearization, and uses hash sets to efficiently check whether a state has been visited before. Alternatively, JPF enables the user to customize the stateful search by specifying which part(s) of the state to be stored and used for state comparison and making the search backtrack on user-specified conditions.

Model Java Interface (MJI) is a mechanism in JPF that enables accessing the JPF special state representation from the host JVM and hence allows parts of JPF execution to be delegated from JPF into the host JVM. MJI is analogous to the Java Native Interface (JNI) [102] that allows parts of JVM execution to be delegated from the JVM into the native code, written in say the C language. MJI, like JNI, splits executions at the method granularity; specifically, each method can be marked to be executed either in JPF or in the host JVM. (JPF uses special name mangling to mark methods for the host JVM execution.) MJI also provides API that allows the host JVM execution to manipulate the JPF special state representation, for example to read or write field values or to create new objects. One advantage of MJI is that it can be used to improve the performance of JPF [26].

4.4.2 Enabling JPF to Explore the State Space of the J-Sim Simulation Model of AODV

JPF could *not* execute the code for the entire J-Sim simulator and the AODV protocol. This is because of the complexity of the J-Sim simulation model of AODV and because the built-in state linearization algorithm takes into account the entire heap, not just the parts of the state that are relevant to AODV. Therefore, we had to create a simplified version of the J-Sim simulation model of AODV. This version does not have the full generality of the J-Sim simulator but provides the basic functionality needed to run AODV. Following that, we wrote a test driver for the (simplified) J-Sim simulation model of AODV. The driver produces an environment that checks which events are enabled in each state being explored, and initiates the execution of all sequences of enabled events up to a configurable maximum depth ($MAXDEPTH$). Similar drivers were previously used in several studies on JPF [112–114]. Similar to the state space exploration framework in J-Sim, the JPF driver implements an explicit-state stateful search that avoids visiting a state s_1 if another state s_2 has already been visited before and either s_2 simulates s_1 (Section 3.3.1) or the hash code of s_2 is equal to the hash code of s_1. The choice between using either the simulation relation or the equality of the hash codes is up to the user. In what follows, we explain the JPF driver assuming the simulation relation is used.

Figure 4.1 gives the pseudo-code of the driver JPFDriver() for state space exploration in JPF. JPFDriver() is executed in JPF's backtrackable JVM. First, the *aodvNodes* array, whose elements correspond to the wireless nodes in the ad-hoc network, is created and initialized (Figure 4.1, line 3). In order to denote that the initial state has been visited, JPFDriver() has to add the initial state to the *AlreadyVisited* data structure. We keep *AlreadyVisited* in the host JVM; otherwise, JPF would consider it part of the program state, represent it using the special representation and restore it on backtracking. Hence, JPFDriver() adds the initial state to *AlreadyVisited* by invoking the *addInitialState()* function using the MJI API (Figure 4.1, line 4). Following that, we use the MJI API to read the JPF special state representation and construct a global state that corresponds to the initial state in the host JVM. This is achieved by the *constructGlobalState()* function (Figure 4.2, line 2). The implementation of *constructGlobalState()* required a lot of programming effort and time as it requires understanding of the JPF special state representation and the MJI API.

As long as the assertion has not been violated and $MAXDEPTH$ bound has not been reached, the while loop in Figure 4.1 (lines 6 to 17) is executed. Each iteration of the loop corresponds to a state being explored (i.e., *currentState*). The events that are enabled in *currentState* are determined

```
 1. procedure JPFDriver()
 2.    depth = 0 ;
 3.    initialize(aodvNodes) ;
 4.    addInitialState(aodvNodes) ; /* addInitialState() is invoked using MJI API. See Figure 4.2. */
 5.    checkProperty = true ;
 6.    while ( (checkProperty == true) AND (depth < MAXDEPTH) ) {
 7.       EnabledEvents = GenerateEnabledEvents() ;
 8.       eventID = Verify.random(EnabledEvents.size() - 1) ;
 9.       ExecuteEvent(EnabledEvents.get(eventID)) ;
10.       checkProperty = verifyAssertion() ;
11.       if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
12.          Print("Counterexample ") ;
13.          printCounterexample() ;
14.          exit ;
         } /* end if */
15.       else if ( (checkProperty == true) AND ((depth + 1) < MAXDEPTH) )
16.          Verify.ignoreIf(wasSimulated(aodvNodes)) ; // wasSimulated() is invoked using MJI API. See Figure 4.2.
17.       depth = depth + 1 ;
      } /* end while */

18. EventInfoList GenerateEnabledEvents()
19.    EnabledEvents = { } ;
20.    for ( all protocol entities p ) { /* for all protocol entities */
21.       for ( all possible events e ) { /* for all events */
22.          NumberOfNextStates = EnablingFunction(p, e) ;
23.          for ( int i = 0 ; i < NumberOfNextStates ; i++ )  /* for all integer-valued parameters */
24.             EnabledEvents.add(new EventInfo(p, e, i)) ;
         } /* end for */
      } /* end for */
25.    return EnabledEvents ;
```

Figure 4.1: JPF driver for state space exploration: Code executed in JPF's backtrackable JVM.

```
 1. native void addInitialState(MJIEnv env, int classRef, int objRef)
    /* addInitialState() is ‘‘native’’; hence, it is executed in the regular host JVM. */
 2.    initialState = constructGlobalState(env, objRef) ;
 3.    AlreadyVisited.add(initialState) ;

 4. native boolean wasSimulated(MJIEnv env, int classRef, int objRef)
    /* wasSimulated() is ‘‘native’’; hence, it is executed in the regular host JVM. */
 5.    nextState = constructGlobalState(env, objRef) ;
 6.    if ( nextState is not simulated by any state in AlreadyVisited ) { /* use protocol-specific properties */
 7.       if ( search strategy is BeFS )
 8.          nextState.computeBeFSTuple() ;  /* use protocol-specific properties */
 9.       AlreadyVisited.add(nextState) ;
10.       return false ;
      }
11.    else
12.       return true ;
```

Figure 4.2: JPF driver for state space exploration: Code executed in the regular host JVM. *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation.

by invoking the *GenerateEnabledEvents()* function (Figure 4.1, line 7). These events are stored in the *EnabledEvents* list, which is returned by *GenerateEnabledEvents()* (Figure 4.1, line 25).

Following that, the JPF's library method Verify.random(int maxBound) is invoked (Figure 4.1, line 8); this method nondeterministically returns a number between zero and $maxBound$ inclusive. Note that Verify.random(int maxBound) does not produce a single execution corresponding to a single (random) number between zero and $maxBound$. Instead, Verify.random(int maxBound) produces $(maxBound + 1)$ executions corresponding to the $(maxBound + 1)$ different values between zero and $maxBound$. Hence, Figure 4.1, line 8, nondeterministically chooses an enabled event. This chosen event is then executed (Figure 4.1, line 9) and the new state's information is stored in the

aodvNodes array. JPFDriver() then checks the assertion (Figure 4.1, line 10). We use the same definitions of the *safe* and *unsafe* events that we used in Section 3.3.2. Furthermore, we have also implemented the *DoesCounterexampleContainEvent()* function (Section 3.3.2) that allows the user to specify that a counterexample has to contain at least one state that is generated due to a particular event.

If the assertion is violated (i.e., the case of an unsafe event) and *DoesCounterexampleContainEvent()* returns true, a counterexample is printed (Figure 4.1, line 13). On the other hand, if the assertion is not violated (i.e., the case of a safe event) and the depth of the next state in the state space is strictly less than the specified maximum depth $MAXDEPTH$ (Figure 4.1, line 15), JPFDriver() has to check whether the next state is simulated by a state that has been visited before. Since the *AlreadyVisited* data structure is kept in the host JVM, JPFDriver() has to invoke the *wasSimulated()* function using the MJI API (Figure 4.1, line 16). If *wasSimulated()* returns true (i.e., the case of a safe non-tree event), the JPF's library method Verify.ignoreIf() instructs JPF to backtrack the execution; hence, another enabled event will be nondeterministically chosen in Figure 4.1, line 8. If not (i.e., the case of a safe tree event), the state space exploration proceeds in the search order determined by the search strategy: breadth-first (BFS), depth-first (DFS) or best-first (BeFS). It should be noted that the driver (Figure 4.1) does not contain code for storing and restoring states because this is done by JPF. (In our state space exploration framework, storing and restoring states is implemented by adding states to, and removing them from, *ToBeExplored*.)

In summary, the tasks that are executed inside the JPF's backtrackable JVM are: the overall state space exploration process, checking which events are enabled in each state being explored, executing the enabled events and checking the assertion. On the other hand, the tasks that are executed inside the host JVM are: implementing the stateful search (using *AlreadyVisited*), and implementing a best-first search (using state ranking by assigning a tuple $< b_1, b_2, \ldots, b_{B-1}, b_B >$ to each state being generated).

The reasons why the stateful search has to be done inside the host JVM instead of the JPF's backtrackable JVM are that we do not want JPF to consider *AlreadyVisited* as part of the program state as explained above, and that the stateful search using the simulation relation with the previously visited states depends on the *protocol-specific* properties (e.g., the routing table entries in AODV and the AODV messages). Due to the JPF special state representation, these protocol-specific properties are represented using (arrays of) integers; hence, not directly accessible. Therefore, we make use of the MJI API to manipulate the JPF special state representation, extract the protocol-specific

properties and construct an instance of the *GlobalState* class inside the regular host JVM. This is achieved by the *constructGlobalState()* function call (Figure 4.2, line 2 and line 5). After constructing the global state, we can check state similarity with the previously visited states (Figure 4.2, line 6) and add the global state to *AlreadyVisited* if necessary (Figure 4.2, line 3 and line 9).

Using state ranking to enable a best-first search strategy is achieved by assigning a BeFS tuple $< b_1, b_2, \ldots, b_{B-1}, b_B >$ to *nextState* (Figure 4.2, line 8) in the same way as we did with instances of the *GlobalState* class in J-Sim. The reason why we choose to implement the computation of the BeFS tuple inside the host JVM is only for programming ease and efficiency since we have to construct the global state inside the host JVM anyway.

4.4.3 Results of the Comparison between the J-Sim State Space Exploration Framework and JPF

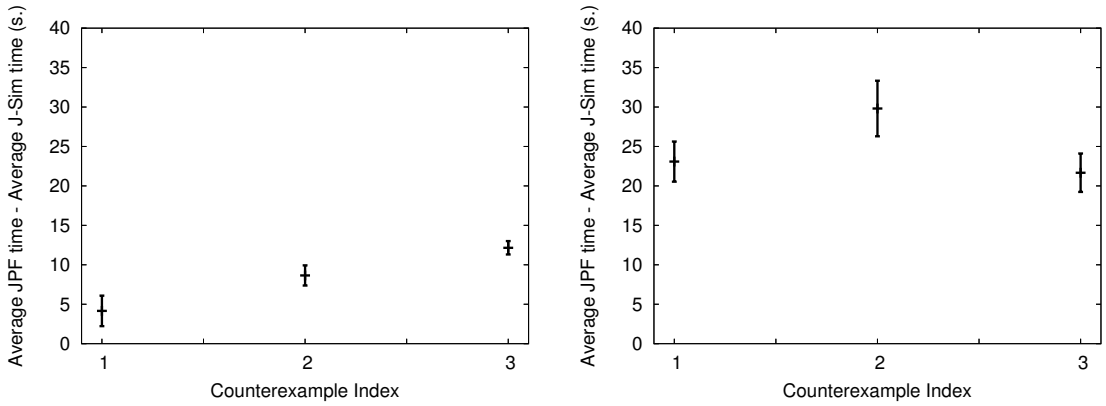
Table 4.2 gives the (i) time, (ii) space (size of *AlreadyVisited*), and (iii) number of events executed for finding the three counterexamples using both J-Sim and JPF with several search strategies. In Table 4.2, *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation. Since we used the same definition of the simulation relation that determines state similarity and the same implementations of BeFS heuristics for state ranking in both tools, the number of events executed and the number of states in *AlreadyVisited* are *exactly* the same for both J-Sim and JPF. (We have also verified that this is the case for lower values of $MAXDEPTH$ where the assertion violations do not occur. In fact, we have also verified that the *sequence* in which states are visited is exactly the same for both J-Sim and JPF.) Therefore, both J-Sim and JPF are having the same amount of "workload". Table 4.2 shows the time needed by both tools to finish that workload. The last column of Table 4.2 shows the ratio between the time needed by JPF and that needed by J-Sim to find the assertion violation. In the case that a small number of events is executed (e.g., the cases of AODV-2-BeFS-AN and AODV-3-BeFS-AN), JPF is much slower than J-Sim. In the case that a moderate number of events is executed (e.g., the cases of AODV-4-BeFS-AN and AODV-5-BeFS-AN in Counterexamples 1 and 3), JPF is slower than J-Sim. In the case that a large number of events is executed (e.g., the cases of DFS-R and BFS-AN), the time needed to find an assertion violation by JPF is close to that of our state space exploration framework in J-Sim, with JPF getting close to, or outperforming, J-Sim in the cases of BFS-AN only.

Counterexample 1 $MAXDEPTH = 10$	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	4699	26303	103.445	111.371	0.929
DFS-R	1459	21757	41.249	9.632	4.282
AODV-1-BeFS-AN	489	4169	8.988	1.012	8.881
AODV-2-BeFS-AN	394	1606	5.289	0.545	9.705
AODV-3-BeFS-AN	240	1179	4.085	0.363	11.253
AODV-4-BeFS-AN	745	8229	16.865	2.225	7.580
AODV-5-BeFS-AN	588	6025	12.679	1.531	8.282
Counterexample 2 $MAXDEPTH = 10$	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	4762	26641	105.282	113.905	0.924
DFS-R	1502	22364	41.954	10.074	4.165
AODV-1-BeFS-AN	485	4116	9.034	1.000	9.034
AODV-2-BeFS-AN	389	1562	5.227	0.529	9.881
AODV-3-BeFS-AN	240	1140	4.041	0.355	11.383
AODV-4-BeFS-AN	2123	25205	49.614	13.507	3.673
AODV-5-BeFS-AN	1809	20976	41.507	10.412	3.986
Counterexample 3 $MAXDEPTH = 10$	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	4310	25211	94.571	97.083	0.974
DFS-R	1406	21056	39.703	9.101	4.362
AODV-1-BeFS-AN	477	4039	8.856	0.972	9.111
AODV-2-BeFS-AN	386	1552	5.202	0.516	10.081
AODV-3-BeFS-AN	239	1131	4.022	0.344	11.692
AODV-4-BeFS-AN	743	8216	16.902	2.196	7.697
AODV-5-BeFS-AN	581	5806	12.479	1.473	8.472

Table 4.2: AODV case study: Time and space requirements (size of *AlreadyVisited*) and the number of events executed for finding the three counterexamples in a 3-node chain ad-hoc network using both J-Sim and JPF with several search strategies. *AlreadyVisited* stores the states that have already been visited. The stateful search depends on the simulation relation.

In order to further compare the performance of the state space exploration framework in J-Sim with that in JPF in the cases that a large number of events is executed, we implemented the shuffling algorithm, which we mentioned in Section 3.3.4, in JPF and ran 100 experiments for each of the BFS-ANS and DFS-RS search strategies and the three counterexamples. Each experiment has a different seed, but the same set of 100 seeds were used for each of J-Sim and JPF. In all experiments, we have verified that the sequence in which states are visited is exactly the same for both J-Sim and JPF; i.e., the common random numbers are synchronized. Figure 4.3 shows the difference between the average JPF time and the average J-Sim time for finding the three counterexamples in a 3-node chain ad-hoc network topology. In Figure 4.3, *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation. As shown in Figure 4.3, the time needed to find an assertion violation by our state space exploration framework in J-Sim can be several seconds less than that of JPF.

Next, we compare the scalability of our state space exploration framework in J-Sim with that of JPF. Specifically, we study the effect of increasing the number of nodes, N, in the network on the performance of both tools with respect to the time needed to find an assertion violation. Table 4.3 gives the (i) time, (ii) space (size of *AlreadyVisited*), and (iii) number of events executed for finding Counterexample 3 in a N-node chain ad-hoc network using both J-Sim and JPF with the AODV-



(a) BFS-ANS (b) DFS-RS

Figure 4.3: AODV case study: The difference between the average JPF time and the average J-Sim time for finding the three counterexamples in a 3-node chain ad-hoc network topology using the BFS-ANS and DFS-RS search strategies and $MAXDEPTH = 10$. We report the difference in means obtained from the 100 experiments and the 99% confidence interval. *AlreadyVisited* stores the states that have already been visited. The stateful search depends on the simulation relation.

N	$MAXDEPTH$	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
3	15	74	155	2.278	0.138	16.507
4	20	402	2235	8.303	0.789	10.523
5	25	1332	12976	48.327	8.323	5.806
6	30	548	3447	19.009	1.514	12.555
7	35	628	2605	22.633	1.793	12.623
8	40	838	2931	36.588	3.154	11.601
9	45	1578	23658	159.922	22.371	7.149
10	50	3875	118119	873.809	247.595	3.529
11	55	4294	143401	1161.203	342.591	3.389
12	60	6302	257045	2417.541	834.982	2.895

Table 4.3: AODV case study: Time and space requirements (size of *AlreadyVisited*) and the number of events executed for finding Counterexample 3 in a N-node chain ad-hoc network using both J-Sim and JPF with the AODV-1-BeFS-AN search strategy. *AlreadyVisited* stores the states that have already been visited. The stateful search depends on the simulation relation.

1-BeFS-AN search strategy. In Table 4.3, *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation. As shown in Table 4.3, our framework is able to find a counterexample in larger network topologies within reasonable time and space requirements. Furthermore, in the case that a large number of events is executed (e.g., the cases of $N = 11$ and $N = 12$), the time needed to find an assertion violation in JPF is slightly larger than that in our state space exploration framework in J-Sim.

The overall conclusion drawn from Tables 4.2-4.3 and Figure 4.3 is that our state space exploration framework in J-Sim is slightly faster than JPF in terms of the time needed to find an assertion violation. The reason why J-Sim is only *slightly* faster than JPF is that the operation that takes the

Operation	JPF Time (s.)	J-Sim Time (s.)
Generate enabled events	18.298	0.095
Shuffle enabled events	3.853	0.013
Execute event handlers	10.020	1.011
Compute a hash code	0.000	0.000
Verify an assertion	2.900	0.096
Construct global state	1.627	0.000
Search in $AlreadyVisited$	37.069	85.133
Insert in $AlreadyVisited$	0.013	0.229
Subtotal	73.780	86.577
Other operations	17.880	0.924
Total time	91.660	87.501
JPF/J-Sim time ratio		1.050
Number of executed events	22946	22946

Table 4.4: AODV case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations in J-Sim and JPF for a case in which a large number of events is executed. Specifically, the example shown corresponds to the following scenario: a 3-node chain ad-hoc network, Counterexample 1, $MAXDEPTH = 10$, search strategy is BFS-ANS and the number of replications is 100. *AlreadyVisited* stores the states that have already been visited. The stateful search depends on the simulation relation.

most significant portion of the state space exploration time is the search in *AlreadyVisited* for any previously visited state that simulates the state that has just been generated (Figure 3.2, line 18 and Figure 4.2, line 6). Since this operation is executed in the regular JVM in both tools, the overhead incurred by JPF's backtrackable JVM due to backtracking and storing and restoring states is only slightly manifested. As an example, we show in Table 4.4 a breakdown of the average state space exploration time spent in some selected operations in J-Sim and JPF for a case in which a large number of events is executed. As shown in Table 4.4, the search in *AlreadyVisited* is the operation that takes the most significant portion of the state space exploration time in both J-Sim and JPF.

In order to show how the overhead incurred by JPF's backtrackable JVM can significantly manifest itself, we let *AlreadyVisited* store the *hash codes* of the states that have already been visited instead of the states themselves. In this setting, there is no need for the *constructGlobalState()* function. Furthermore, the search in *AlreadyVisited* for a hash code that is equal to the hash code of the state that has just been generated becomes an extremely cheap operation that only takes a very small portion of the state space exploration time. On the other hand, the operations of executing the events, computing the hash codes, checking the assertion, determining the enabled events, and computing the BeFS tuple (in the case of a BeFS strategy), which are now all done in JPF's backtrackable JVM, become the operations that take the most significant portion of the state space exploration time. For the sake of completeness, we include the pseudo-code of the JPF driver in this case in Figure 4.4 and the code executed in the host JVM in Figure 4.5. The advantage of this setting is that it requires less programming effort since the user does not need to implement the *constructGlobalState()* function. We still keep *AlreadyVisited* in the regular host JVM since we

```
 1. procedure JPFModifiedDriver()
 2.    depth = 0 ;
 3.    initialize(aodvNodes) ;
 4.    hashCode = computeHashCode(aodvNodes) ;
 5.    addInitialStateHashCode(hashCode) ; // addInitialStateHashCode() is invoked using MJI API. See Figure 4.5.
 6.    checkProperty = true ;
 7.    while ( (checkProperty == true) AND (depth < MAXDEPTH) ) {
 8.       EnabledEvents = GenerateEnabledEvents() ;  /* See Figure 4.1 for GenerateEnabledEvents() */
 9.       eventID = Verify.random(EnabledEvents.size() - 1) ;
10.       ExecuteEvent(EnabledEvents.get(eventID)) ;
11.       hashCode = computeHashCode(aodvNodes) ;
12.       checkProperty = verifyAssertion() ;
13.       if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
14.          Print(``Counterexample `') ;
15.          printCounterexample() ;
16.          exit ;
         } /* end if */
17.       else if ( (checkProperty == true) AND ((depth + 1) < MAXDEPTH) ) {
18.          Verify.ignoreIf(wasVisited(hashCode)) ; // wasVisited() is invoked using MJI API. See Figure 4.5.
19.          if ( search strategy is BeFS )
20.             computeBeFSTuple(aodvNodes) ;  /* use protocol-specific properties */
         } /* end if */
21.       depth = depth + 1 ;
      }  /* end while */
```

Figure 4.4: JPF modified driver for state space exploration: Code executed in JPF's backtrackable JVM.

```
 1. native void addInitialStateHashCode(MJIEnv env, int classRef, int objRef)
    /* addInitialStateHashCode() is ``native''; hence, it is executed in the regular host JVM. */
 2.    hashCode = env.getIntField(objRef) ; /* retrieve the hash code */
 3.    AlreadyVisited.add(hashCode) ;

 4. native boolean wasVisited(MJIEnv env, int classRef, int objRef)
    /* wasVisited() is ``native''; hence, it is executed in the regular host JVM. */
 5.    hashCode = env.getIntField(objRef) ; /* retrieve the hash code */
 6.    if ( hashCode does not exist in AlreadyVisited ) {
 7.       AlreadyVisited.add(hashCode) ;
 8.       return false ;
       }
 9.    else
10.       return true ;
```

Figure 4.5: JPF modified driver for state space exploration: Code executed in the regular host JVM. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

do not want JPF to consider it as part of the program state as explained above.

Tables 4.5 and 4.6 give the performance results of both tools under this setting. As we did above, we used the same hash code computation in both tools. Specifically, the hash code of a state is computed by first constructing an integer array representation of the state. Following that, an integer-valued hash code of this integer array representation is computed using the Jenkins' hash function that is part of the JPF (version 4.1) distribution. (We slightly modified this function to return an integer instead of a long value.) The integer array representation of a state depends on the protocol-specific information such as the AODV packet headers and payloads, and each node's $seqno_n$, bid_n, routing table entries and broadcast ID cache entries. The reason we used this integer array representation instead of the linearization algorithm built-in inside JPF is that the JPF linearization algorithm linearizes the entire heap, but we are only interested in the AODV protocol-

Counterexample 1 $MAXDEPTH = 10$	Space (number of hash codes)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	70308	283122	1607.966	21.880	73.490
DFS-R	22908	344333	1789.221	21.365	83.745
AODV-1-BeFS-AN	1346	10878	50.611	1.173	43.147
AODV-2-BeFS-AN	6491	85977	401.103	5.706	70.295
AODV-3-BeFS-AN	569	2560	12.129	0.484	25.060
AODV-4-BeFS-AN	4431	45816	232.117	3.411	68.050
AODV-5-BeFS-AN	2398	19525	104.798	1.834	57.142
Counterexample 2 $MAXDEPTH = 10$	Space (number of hash codes)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	70551	284689	1611.868	21.957	73.410
DFS-R	23290	350204	1839.625	21.784	84.448
AODV-1-BeFS-AN	1302	10524	48.895	1.138	42.966
AODV-2-BeFS-AN	6488	85941	413.173	5.711	72.347
AODV-3-BeFS-AN	530	2284	10.929	0.457	23.915
AODV-4-BeFS-AN	17004	183929	970.930	12.129	80.050
AODV-5-BeFS-AN	15147	159198	844.428	10.791	78.253
Counterexample 3 $MAXDEPTH = 10$	Space (number of hash codes)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	69073	280967	1579.651	21.552	73.295
DFS-R	22767	342585	1793.470	21.286	84.256
AODV-1-BeFS-AN	1291	10414	47.722	1.106	43.148
AODV-2-BeFS-AN	6473	85779	410.389	5.670	72.379
AODV-3-BeFS-AN	529	2275	10.853	0.442	24.554
AODV-4-BeFS-AN	4427	45816	236.260	3.401	69.468
AODV-5-BeFS-AN	2384	19166	101.402	1.796	56.460

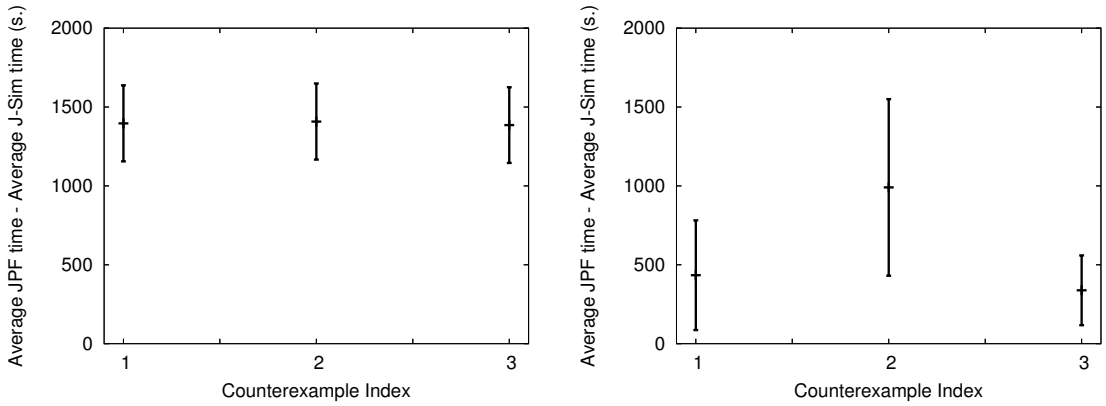
Table 4.5: AODV case study: Time and space requirements (size of *AlreadyVisited*) and the number of events executed for finding the three counterexamples in a 3-node chain ad-hoc network using both J-Sim and JPF with several search strategies. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

N	$MAXDEPTH$	Space (number of hash codes)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
3	15	86	134	2.667	0.133	20.053
4	20	520	1988	13.300	0.462	28.788
5	25	7094	62206	495.295	5.897	83.991
6	30	3437	42631	404.387	4.610	87.720
7	35	5269	102833	1242.740	11.185	111.108

Table 4.6: AODV case study: Time and space requirements (size of *AlreadyVisited*) and the number of events executed for finding Counterexample 3 in a N-node chain ad-hoc network using both J-Sim and JPF with the AODV-1-BeFS-AN search strategy. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

specific information in order to ensure a fair comparison between J-Sim and JPF. Furthermore, we used the same implementations of BeFS heuristics for state ranking in both J-Sim and JPF. Hence, the number of events executed and the number of hash codes in *AlreadyVisited* are *exactly* the same for both tools. (We have also verified that this is the case for lower values of $MAXDEPTH$ where the assertion violations do not occur and that the *sequence* in which states are visited is exactly the same for both J-Sim and JPF.) As shown in Tables 4.5 and 4.6, our state space exploration framework in J-Sim can be up to almost two orders of magnitude faster than that in JPF.

In order to further compare the performance of the state space exploration framework in J-Sim with that in JPF in this setting, we ran 20 experiments for each of the BFS-ANS and DFS-RS search strategies and the three counterexamples in both J-Sim and JPF. Each experiment has a different



(a) BFS-ANS

(b) DFS-RS

Figure 4.6: AODV case study: The difference between the average JPF time and the average J-Sim time for finding the three counterexamples in a 3-node chain ad-hoc network topology using the BFS-ANS and DFS-RS search strategies and $MAXDEPTH = 10$. We report the difference in means obtained from the 20 experiments and the 99% confidence interval. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

seed, but the same set of 20 seeds were used for each of J-Sim and JPF. Similar to what we did above, we have verified that the sequence in which states are visited is exactly the same for both J-Sim and JPF in all experiments. Figure 4.6 shows the difference between the average JPF time and the average J-Sim time for finding the three counterexamples in a 3-node chain ad-hoc network topology. In Figure 4.6, *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes. As shown in Figure 4.6, the time needed to find an assertion violation by our state space exploration framework in J-Sim is significantly less than that of JPF.

In order to understand the reasons behind this observation, we show in Table 4.7 a breakdown of the average state space exploration time spent in some selected operations in J-Sim and JPF for a case in which a large number of events is executed. As shown in Table 4.7, the operations of executing the events, computing the hash codes, checking the assertion, and determining the enabled events, which are all done in JPF's backtrackable JVM, take more than 81% of the state space exploration time in JPF. On the other hand, the corresponding operations in J-Sim take much less time and thus make the state space exploration framework in J-Sim significantly faster than that in JPF.

Operation	JPF Time (s.)	J-Sim Time (s.)
Generate enabled events	59.960	0.540
Shuffle enabled events	12.740	0.065
Execute event handlers	84.448	4.388
Compute a hash code	652.444	1.762
Verify an assertion	22.510	0.690
Construct global state	0.000	0.000
Search in $AlreadyVisited$	0.184	0.111
Insert in $AlreadyVisited$	0.059	0.104
Subtotal	832.345	7.660
Other operations	170.240	4.550
Total time	1002.585	12.210
JPF/J-Sim time ratio		82.110
Number of executed events	192312	192312

Table 4.7: AODV case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations in J-Sim and JPF for another case in which a large number of events is executed. Specifically, the example shown corresponds to the following scenario: a 3-node chain ad-hoc network, Counterexample 2, $MAXDEPTH = 10$, search strategy is DFS-RS and the number of replications is 20. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

4.5 Lessons Learned

The results presented in this chapter show that our state space exploration framework in J-Sim can be significantly faster than that in JPF (Figure 4.6) unless a significant amount of programming effort is done in JPF to make its performance close to that of our SSE framework in J-Sim (Figure 4.3). This result justifies the need for building a framework in J-Sim for the verification of network simulation models by state space exploration instead of using a general-purpose model checking tool since we believe that wireless network protocol designers and simulation modelers will feel more comfortable using J-Sim as a single integrated environment for both building a simulation model and verifying its correctness than using J-Sim for building a simulation model and using another tool (JPF) for verifying its correctness.

Chapter 5

Directed Diffusion Case Study

In this chapter, we apply the J-Sim state space exploration framework to the J-Sim simulation model of the Directed Diffusion data dissemination protocol for wireless sensor networks. In Section 5.1, we give an overview of the key functionality of directed diffusion. In Section 5.2, we describe the steps that we follow to verify its simulation model including how we exploit protocol-specific properties. In Section 5.3, we present the results of this verification. Finally, in Section 5.4, we summarize the lessons that we learned in the two case studies of AODV and directed diffusion.

The J-Sim simulation model of directed diffusion (not including the J-Sim library) has about 1400 lines of code. We conduct all the experiments on a dual-processor Intel Xeon 2.8 GHz machine running Linux version 2.6.17 with 2 GB memory. We use Sun's 1.5.0 04-b05 Java HotSpot^{TM} Client VM with 0.5 GB initial heap size and 1.5 GB maximum heap size.

5.1 Overview of Directed Diffusion

A major objective of a wireless sensor network (WSN) is to monitor and sense events of interests (e.g., changes in the acoustic sound, seismic, or temperature) in a specific environment. Events of interest are generated by *target nodes*. For instance, a moving tank in a battlefield may generate ground vibrations that can be detected by seismic sensors. Upon detecting an event of interest, *sensor nodes* send reports to *sink (user) nodes* either periodically or on demand. From the perspective of network simulation, a WSN typically consists of these three types of nodes: sensor nodes (that sense and detect the events of interest), target nodes (that generate events of interest), and sink nodes (that utilize and consume the sensor information). The implementation details of the simulation and emulation frameworks for WSNs in J-Sim can be found in [88, 89, 91]. In this section, we describe the J-Sim simulation model of directed diffusion.

Directed diffusion [48] is a *data-centric* information dissemination paradigm for WSNs. Conceptually, *data* in WSNs is the collected (or processed) information of a physical phenomenon. In

directed diffusion, a sink node periodically broadcasts an INTEREST packet, containing the description of a sensing task that it is interested in (e.g., detecting a chemical explosion in a specific area). INTEREST packets are diffused throughout the network; e.g., via flooding. After receiving an INTEREST packet, a node may decide to re-send the INTEREST packet to its neighbors, or suppress a received INTEREST if it has recently resent a matching INTEREST. INTEREST packets are used to set up *exploratory gradients*. A gradient is the direction state created in each node that receives an INTEREST, where the gradient direction is set toward the neighboring node from which the INTEREST is received. It should be noted that this mechanism results in neighboring nodes establishing a gradient toward each other. This is because when a node receives an INTEREST from its neighbor, it has no way of knowing whether that INTEREST was in response to one it sent out earlier or is an identical INTEREST from another sink on the other side of that neighbor.

Each node maintains an interest cache. Each interest entry in this cache corresponds to a distinct interest and stores information about the gradients that a node has (up to one gradient per neighbor) for that interest. Each gradient in an interest entry has a lifetime that is determined by the sink node. When a gradient expires, it is removed from its interest entry. When all gradients in an interest entry have been removed, the interest entry itself is removed from the interest cache.

When an INTEREST packet arrives at a sensor node that can sense data which matches the interest, this sensor node becomes a *source node*, prepares DATA packets (each of which describes the sensed data), and sends them to neighbors for whom it has a gradient once every *exploratory interval*. Each node also maintains a data cache that keeps track of recently seen DATA packets. When a node receives a DATA packet, if the DATA packet has a matching data cache entry, it is discarded; otherwise, the node adds the received DATA packet to the data cache and forwards it to each neighbor for whom the node has a gradient. As a result, DATA packets are forwarded toward the sink node(s) along (possibly) multiple gradient paths.

Upon receipt of a DATA packet, a sink node *reinforces* its preferred neighbor based on a data-driven local rule. For instance, the sink node may reinforce any neighbor from which it received previously unseen data (i.e., the neighbor from which it first received the latest data matching the interest). The data cache is used to determine that preferred neighbor. In order to reinforce a neighbor, the sink node sends a *positive reinforcement* packet to that neighbor to inform it of sending data at a smaller interval (i.e., higher rate) than the exploratory interval, thereby establishing a *reinforced gradient* (also called *data gradient*) towards the sink node. The reinforced neighbor will in turn reinforce its preferred neighbor. This process repeats all the way back to the data source,

resulting in a reinforced path (i.e., a chain of reinforced gradients) between the source and the sink nodes. It should be noticed that this mechanism can result in more than one path being reinforced, thereby consuming more energy. Furthermore, one reinforced path may turn out to be consistently "better" than another path, which then needs to be negatively reinforced. Specifically, a *negative reinforcement* packet is used to inform a neighbor to send data at the lower rate determined by the exploratory interval. Similar to positive reinforcements, a data-driven local rule is used to decide whether to negatively reinforce a neighbor or not. One plausible rule is to negatively reinforce a neighbor from whom no new events have been received within a window of W events (i.e., the neighbor that consistently sends previously seen events).

5.2 Verifying the Simulation Model of Directed Diffusion

We follow the same steps in Section 4.2 to verify the simulation model of directed diffusion.

Step 1. States: Definitions of the global state, the initial state, and the assertion

We use the same definitions of *GlobalState* and network cloud that were introduced in Section 4.2. On the other hand, since the protocol state is protocol-specific, the protocol state in directed diffusion includes each node's interest cache and data cache. In the initial global state, the network does not contain any packets and the directed diffusion process at each node starts with an empty interest cache and an empty data cache.

The assertion that we check is the *loop-free* property of the reinforced path. Consider two nodes n and m where $RPath(n, m)$ is true if and only if there is a reinforced path from n to m. The loop-free property can be expressed as follows:

$$\neg \ (\ RPath(n, m)\ \wedge\ RPath(m, n)\ )$$

Step 2. Events

Next, we specify the set of events, when each event is enabled and the corresponding *EnablingFunction()*, and how each event is handled. We classify the events into two categories: node events and network events. The events in each category are listed as follows:

1. Node Events

T_0 Initiation of a sensing task by node n: This event is enabled if node n is a sink node. When enabled, *EnablingFunction(currentState, n, T_0)* returns 1. The event is handled by broadcasting an INTEREST packet.

T_1 Restart of the directed diffusion process at node n: This event may take place because of a node reboot. This event is always enabled; i.e., *EnablingFunction(currentState, n, T_1)* always returns 1. The event is handled by reinitializing the state of the directed diffusion process at node n.

T_2 Gradient timeout at node n: This event is enabled if the interest cache of node n contains at least one interest entry that has at least one gradient. When enabled, *EnablingFunction(currentState, n, T_2)* returns the total number of gradients in the interest cache of node n. The event is handled by removing one of the gradients in the interest cache of node n. If all gradients in an interest entry have been removed, the interest entry itself is removed from the interest cache.

T_3 Data cache timeout[1] at node n: This event is enabled if there is at least one entry in the data cache of node n. When enabled, *EnablingFunction(currentState, n, T_3)* returns the number of entries in the data cache of node n. The event is handled by deleting an entry from the data cache of node n.

2. Network Events

T_4 Delivering a packet to node n: This event is enabled if the network contains at least one packet that is destined for node n such that node n is one of the neighbors of the source of the packet. When enabled, *EnablingFunction(currentState, n, T_4)* returns the number of the packets that satisfy this condition. The event is handled by removing one of these packets from the network and forwarding it to node n in order to be processed as explained in Section 5.1.

T_5 Loss of a packet destined for node n: This event is enabled if the network contains at least one packet that is destined for node n. When enabled, *EnablingFunction(currentState, n, T_5)* returns the number of the packets that satisfy this condition. The event is handled by removing one of these packets from the network.

[1]For practical reasons, previously received DATA packets can not be kept in the data cache for an indefinitely long time; otherwise, the size of the data cache can increase arbitrarily. In the J-Sim simulation model of directed diffusion, each DATA packet in the data cache is associated with a lifetime. Periodically, a data cache timeout event is triggered causing the deletion of entries in the cache that have expired.

Step 3. Simulation Relation

Again, we use the general simulation relation outlined in the introduction and Section 3.3. For directed diffusion this reduces to the following definition. A state s_2 is said to simulate a state s_1 if (i) s_1 and s_2 have the same neighborhood information, (ii) for each packet in s_1, there is a corresponding equivalent packet in s_2, and (iii) for each node n, s_1 and s_2 have correspondingly equal node n's interest cache and data cache (each viewed as an unordered set of entries).

In Appendix A, we argue that this relation is a simulation relation.

Step 4. State Ranking: Exploiting protocol-specific properties

Recall from Chapter 3 that in order to enable a best-first search strategy, the user needs to write a Java method that maps each state to a tuple $< b_1, b_2, \ldots, b_{B-1}, b_B >$ based on protocol-specific properties such that a state s_1 is considered "better than" a state s_2 if s_1 has a higher lexicographical order of this tuple than s_2.

In the course of verifying the J-Sim simulation model of AODV (Section 4.3), AODV-1-BeFS and AODV-3-BeFS provided comparatively better performance results. Hence we use these two BeFS heuristics to devise two corresponding BeFS heuristics for directed diffusion. In particular, the loop-free property for AODV involves only valid RTEs to a destination d, and by analogy, the loop-free property for directed diffusion involves only reinforced gradients. Similarly, data packets are forwarded in AODV based on the next hop information stored in the valid RTEs, and by analogy, data packets are forwarded in directed diffusion based on the gradients established at the nodes. Therefore, two potentially good BeFS heuristics for exploring the state space of directed diffusion are:

1. DD-1-BeFS: This heuristic assigns to a state s a BeFS tuple that consists of one component $< b_1 >$ such that b_1 is the total number of *both exploratory and reinforced gradients* in s.

2. DD-2-BeFS: This heuristic assigns to a state s a BeFS tuple that consists of two components $< b_1, b_2 >$ such that b_1 is the number of *reinforced* gradients in s, and b_2 is the total number of *both exploratory and reinforced* gradients in s.

Along a similar line of arguments, we also devise the following BeFS heuristics:

1. DD-3-BeFS: Since a reinforced gradient is established upon receiving a positive reinforcement packet, DD-3-BeFS assigns to a state s a BeFS tuple that consists of one component $< b_1 >$ such that b_1 is the number of positive reinforcement packets in s.

2. DD-4-BeFS: DD-4-BeFS assigns to a state s a BeFS tuple that consists of two components $< b_1, b_2 >$ such that b_1 is the number of positive reinforcement packets in s (as in DD-3-BeFS), and b_2 is the total number of *both exploratory and reinforced* gradients in s.

3. DD-5-BeFS: This heuristic assigns to a state s a BeFS tuple that consists of one component $< b_1 >$ such that b_1 is the total number of data cache entries at all nodes in s.

4. DD-6-BeFS: DD-6-BeFS assigns to a state s a BeFS tuple that consists of two components $< b_1, b_2 >$ such that b_1 is the total number of data cache entries at all nodes in s (as in DD-5-BeFS), and b_2 is the total number of *both exploratory and reinforced* gradients in s.

5.3 Results of the Verification

We consider an initial state that consists of a chain topology of N nodes: n_0 (the only sink node, which is interested in a single sensing task), n_1, n_2, ..., n_{N-2}, and n_{N-1} (the only source node, which can sense data that matches the interest). A loop in the reinforced path may take place because the interest and gradient setup mechanisms themselves do *not* guarantee loop-free reinforced paths between the source and the sink nodes. In order to prevent loops from taking place, the data cache is used to suppress previously seen DATA packets. However, we discover that, in the case of (a) the deletion of a DATA packet from the data cache and/or (b) a node reboot (which effectively deletes all the entries in the data and interest caches), a loop may be created. The loop that is created in the first case is referred to as Counterexample 1 while the loop that is created in the second case is referred to as Counterexample 2. For instance, consider a chain topology consisting of $N = 4$ nodes. If n_1 accepts a DATA packet sent by n_2, n_2 becomes n_1's preferred neighbor. Now, if n_2 deletes the DATA packet from its data cache due to a data cache timeout (Counterexample 1) or a node reboot (Counterexample 2), it may later accept the DATA packet sent by n_1 (because it will be previously unseen data) causing n_1 to become n_2's preferred neighbor. (Recall that neighboring nodes establish gradients toward each other.) Therefore, n_1 and n_2 may positively reinforce each other causing a loop in the reinforced path. In fact, positive reinforcement packets may not eventually reach the source node causing a disruption in the reinforced path (i.e., the reinforced path may include a loop that does not include the source node).[2] The interested reader is referred to Appendix C for detailed traces (along with the explanations) of the two counterexamples. It has to be mentioned

[2]For Counterexample 2, we require that the counterexample contain at least one state that is generated due to a node reboot event, T_1. Furthermore, in order to show that a loop in the reinforced path may still take place even if the data cache timeout event, T_3, does not happen (i.e., the data cache size is infinite), we disabled T_3; i.e., we made *EnablingFunction(currentState, n, T_3)* always return zero.

	Counterexample 1, $MAXDEPTH = 15$		
	Time (s.)	Space (number of states)	Events
BFS-AN	809.701	16735	115698
DFS-R	319.243	8306	173823
DD-1-BeFS-AN	0.563	513	1042
DD-2-BeFS-AN	0.599	501	1279
DD-3-BeFS-AN	128.526	5638	113342
DD-4-BeFS-AN	0.309	284	474
DD-5-BeFS-AN	542.764	14284	175807
DD-6-BeFS-AN	320.229	12985	95036
	Counterexample 2, $MAXDEPTH = 20$		
	Time (s.)	Space (number of states)	Events
BFS-AN	6410.453	44348	425181
DFS-R	262.659	8027	179112
DD-1-BeFS-AN	306.545	18063	92820
DD-2-BeFS-AN	469.543	20435	132382
DD-3-BeFS-AN	2059.233	21670	521326
DD-4-BeFS-AN	21.882	3574	18373
DD-5-BeFS-AN	N/A	N/A	N/A
DD-6-BeFS-AN	5218.443	55336	508048

Table 5.1: Directed diffusion case study: Time and space requirements (sum of the sizes of *AlreadyVisited* and *ToBeExplored*) and the number of events executed for finding the two counterexamples in a 4-node chain WSN using different search strategies. N/A indicates that the state space explorer is not able to find a counterexample in two hours. *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation.

that although the reinforced path may have a loop, this loop will not continue to exist forever. It will be removed later either by the negative reinforcement mechanism or by the gradient timeout mechanism. Furthermore, forwarding a DATA packet over the loop will stop once all nodes on the loop have received the DATA packet.

Table 5.1 gives the (i) time, (ii) space (sum of the sizes of *AlreadyVisited* and *ToBeExplored*), and (iii) number of events executed for finding the two counterexamples using several search strategies. As shown in Table 5.1, DD-1-BeFS-AN provides orders of magnitude reduction with respect to the evaluation criteria when compared to other standard search strategies such as BFS-AN and DFS-R. Furthermore, DD-4-BeFS-AN outperforms DD-3-BeFS-AN, and DD-6-BeFS-AN outperforms DD-5-BeFS-AN. This is because both DD-4-BeFS and DD-6-BeFS are two-level BeFS heuristics that use DD-1-BeFS if the non-visited states are equally good and are thus able to better guide the best-first search strategy, in the lower depths of the search space, than DD-3-BeFS and DD-5-BeFS respectively.

In order to better study the effect of the choice of the BeFS heuristic, we ran 20 randomized experiments for each BeFS heuristic. Each experiment has a different seed, but the same set of 20 seeds were used for each of the six BeFS heuristics. Table 5.2 reports the minimum, the maximum and the average values for each performance evaluation criterion. As shown in Table 5.2, DD-4-BeFS-ACS and DD-6-BeFS-ACS (DD-4-BeFS-ANS and DD-6-BeFS-ANS) respectively outperform DD-3-BeFS-ACS and DD-5-BeFS-ACS (DD-3-BeFS-ANS and DD-5-BeFS-ANS) in terms of the

Counterexample 1	Time (s.)			Space (number of states)			Events		
$MAXDEPTH$=15	Min	Max	Average	Min	Max	Average	Min	Max	Average
DD-1-BeFS-ACS	0.184	0.512	0.377	204	485	342.00	282	1329	869.60
DD-2-BeFS-ACS	0.176	0.744	0.450	208	481	342.75	233	2261	1193.05
DD-3-BeFS-ACS	143.693	557.510	371.231	6606	12340	9834.05	118106	253210	201005.30
DD-4-BeFS-ACS	0.180	0.368	0.271	198	355	259.30	283	789	501.55
DD-5-BeFS-ACS	155.088	1243.090	699.670	10881	31324	20092.55	113662	395107	267022.60
DD-6-BeFS-ACS	98.702	107.742	102.341	18041	19193	18474.80	96016	100840	97758.20
DD-1-BeFS-ANS	0.236	0.719	0.507	193	631	418.75	282	1231	799.45
DD-2-BeFS-ANS	0.229	1.103	0.599	195	656	407.15	233	2194	1164.00
DD-3-BeFS-ANS	2.939	489.176	209.151	600	10791	6605.30	8319	222677	128465.25
DD-4-BeFS-ANS	0.222	0.530	0.309	193	445	257.25	221	789	407.80
DD-5-BeFS-ANS	246.009	908.874	539.620	8226	19310	12997.50	102115	288012	182186.60
DD-6-BeFS-ANS	322.794	352.894	335.397	13084	14336	13540.45	95698	101561	97930.55
Counterexample 2	Time (s.)			Space (number of states)			Events		
$MAXDEPTH$=20	Min	Max	Average	Min	Max	Average	Min	Max	Average
DD-1-BeFS-ACS	25.197	294.842	160.366	10057	43758	30520.25	46536	200973	139185.10
DD-2-BeFS-ACS	131.078	492.221	238.152	14707	39644	24498.70	107358	255321	160739.35
DD-3-BeFS-ACS	142.102	6373.538	2524.681	5826	46016	24204.90	122964	1050558	573942.50
DD-4-BeFS-ACS	7.140	15.271	12.150	5944	8785	7152.05	17828	35332	29264.55
DD-5-BeFS-ACS	1048.740	7352.659	3246.824	22467	79064	43810.45	340947	1057788	615142.75
DD-6-BeFS-ACS	2336.184	2787.956	2427.339	74870	83479	76540.25	575905	628108	587570.10
DD-1-BeFS-ANS	92.118	581.052	373.419	8357	24759	19406.85	44562	140796	105669.10
DD-2-BeFS-ANS	298.220	786.888	400.639	14680	26383	17972.30	96617	183642	119599.00
DD-3-BeFS-ANS	0.904	4301.194	821.720	295	40114	11919.45	3398	826966	250433.75
DD-4-BeFS-ANS	18.846	28.061	23.176	3415	4275	3709.15	16153	23087	20537.90
DD-5-BeFS-ANS	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DD-6-BeFS-ANS	5314.174	6624.542	5599.007	55258	66900	57866.75	509282	577676	527568.95

Table 5.2: Directed diffusion case study: Time and space requirements (sum of the sizes of *AlreadyVisited* and *ToBeExplored*) and the number of events executed for finding the two counterexamples in a 4-node chain WSN using different randomized best-first search strategies. Number of replications is 20. N/A indicates that in some replications, the state space explorer is not able to find a counterexample in 125 minutes. *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation.

average time to find an assertion violation.

In Table 5.2, N/A indicates that in some replications, the state space explorer is not able to find a counterexample in 125 minutes. This occurred only for the DD-5-BeFS-ANS in Counterexample 2. On the other hand, DD-5-BeFS-ACS was able to find the same counterexample in all replications in less than 125 minutes. We found that this is because in both search strategies the operation that takes the most significant portion of the state space exploration time is the search in *AlreadyVisited* for any previously visited state that simulates *nextState* (in the case of BeFS-ANS) or *currentState* (in the case of BeFS-ACS). Since BeFS-ACS lazily performs this operation as we explained in Section 3.3.3, it is able to find the assertion violation in more replications than BeFS-ANS.

Table 5.3 gives the time and space requirements and the number of events executed for finding Counterexample 1 in a chain topology consisting of N nodes using DD-4-BeFS-AC. Again our framework is able to find a counterexample in larger network topologies within reasonable time and space requirements.

N	MAXDEPTH	Time (s.)	Space (number of states)	Events
4	15	0.252	247	474
5	20	14.667	2289	27826
6	25	38.848	3604	51341
7	30	31.679	4503	51649
8	35	101.222	8506	110540
9	40	205.121	13839	172912
10	45	689.344	24105	347204
11	50	1332.193	33422	495318
12	55	3721.432	51666	867921

Table 5.3: Directed diffusion case study: Time and space requirements (sum of the sizes of *AlreadyVisited* and *ToBeExplored*) and the number of events executed for finding Counterexample 1 in a N-node chain WSN using DD-4-BeFS-AC. *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation.

5.4 Lessons Learned

In this section, we summarize the lessons learned in the two case studies of AODV and directed diffusion. First, we show that the state space exploration framework is able to verify the simulation models of two fairly complex network protocols such as AODV and directed diffusion. This demonstrates that the framework is general enough and not tied to a particular network protocol. To verify the simulation model of another network protocol, one needs to follow the steps as outlined in Section 4.2 and Section 5.2.

Second, we demonstrate that the use of BeFS heuristics reduces the time and space requirements by several orders of magnitude when compared to classic breadth-first and depth-first search strategies. Based on the results obtained for the BeFS heuristics that we devised, we recommend deriving the BeFS heuristic from properties inherent to the network protocol and the assertion being checked. This is justified by the observation that AODV-1-BeFS and DD-1-BeFS provide better performance results in terms of time and space requirements and number of events executed for finding a violation of an assertion as shown in Table 4.1 and Table 5.1 respectively. Furthermore, using a two-level BeFS heuristic, in which a BeFS heuristic such as AODV-1-BeFS or DD-1-BeFS is used if the non-visited states are equally good, also improved the performance. This is justified by the observation that AODV-5-BeFS outperforms AODV-4-BeFS (Table 4.1), DD-4-BeFS outperforms DD-3-BeFS, and DD-6-BeFS outperforms DD-5-BeFS (Table 5.1 and Table 5.2).

Chapter 6

ARQ Case Study

In this chapter, we apply the J-Sim state space exploration framework to the J-Sim simulation model of the Automatic Repeat reQuest (ARQ) protocol. In Section 6.1, we give an overview of ARQ's key functionality. In Section 6.2, we describe the steps that we follow to verify its simulation model including how we exploit protocol-specific properties. In Section 6.3, we present the results of this verification.

The J-Sim simulation model of ARQ (not including the J-Sim library) has about 170 lines of code. We conduct all the experiments on a dual-processor Intel Xeon 2.8 GHz machine running Linux version 2.6.17 with 2 GB memory. We use Sun's 1.5.0 04-b05 Java HotSpot^{TM} Client VM with 0.5 GB initial heap size and 1.5 GB maximum heap size.

6.1 Overview of ARQ

Automatic Repeat reQuest (ARQ) is a well-known error control protocol that has several variations, each of which works between two communicating nodes: a sender and a receiver. ARQ ensures that every data packet sent by the sender will eventually be received correctly by the receiver and that the receiver will get the data packets in order, i.e., it is an in-order reliable unicast protocol.

The simplest form of the ARQ protocol is stop-and-wait ARQ in which the sender sends a single data packet and then waits for a positive acknowledgment (ACK) before it advances to the next data packet. The receiver replies with an ACK if the data packet is correctly received. As either the data packet or the corresponding ACK may be lost/corrupted in transit, the sender sets a retransmission timer after it has sent a data packet. If no ACK is received before the retransmission timer expires, the sender retransmits the data packet. It should be mentioned that the setting of the timeout interval at the sender is very important, and is a trade-off between premature timeout and prolonged retransmission.

For the receiver to distinguish between a data packet that is sent for the first time and a retrans-

mission, a sequence number is included in the header of each data packet. For stop-and-wait ARQ, it is sufficient that the sequence number be 1-bit (i.e., either 0 or 1) because the only ambiguity is between a data packet and its immediate predecessor and successor, but not between the predecessor and successor themselves [100]. For similar reasons, each ACK should also contain a sequence number. In the common practice, the sequence number in the ACK is the sequence number of the next expected data packet rather than the sequence number of the data packet that has been recently received. If the receiver receives a data packet whose sequence number is not equal to the sequence number it is expecting, the receiver discards this duplicate data packet and retransmits an ACK announcing the sequence number of the next expected data packet. Upon receiving the ACK, the sender checks the sequence number in the ACK to determine whether a new data packet or a retransmission should be sent.

6.2 Verifying the Simulation Model of ARQ

We follow the same steps in Sections 4.2-5.2 to verify the simulation model of ARQ.

Step 1. Definitions of the global state, the initial state, and the assertion

We define *GlobalState* as a tuple that has two components, namely the protocol state and the network cloud. The protocol state contains *SeqNoSent* (the sequence number of the data packet that has most recently been sent by the sender and whose ACK the sender is waiting for), *SeqNoExpected* (the sequence number of the data packet that the receiver is expecting), *NumDistinctDataMsgSent* (the total number of distinct data packets sent by the sender), and *NumDistinctDataMsgReceived* (the total number of distinct data packets received by the receiver). The network cloud models the network as an unbounded list that contains the data and ACK packets.

The initial state is the state in which the sender has just sent the first data packet with sequence number 0 (denoted as *D0*), the receiver is expecting *D0* and the network contains *D0*.

An important assertion for any reliable unicast protocol is that the receiver does not miss any data packet that the sender believes to have been received by the receiver. In an ARQ protocol that uses a 1-bit sequence number, this assertion translates to the requirement that the difference between the total number of distinct data packets transmitted by the sender (*NumDistinctDataMsgSent*) and the total number of distinct data packets received by the receiver (*NumDistinctDataMsgReceived*) is always less than or equal to 2. This assertion can be expressed as follows:

$$(NumDistinctDataMsgSent - NumDistinctDataMsgReceived) \leq 2$$

Step 2. Events

Next, we specify the set of events, when each event is enabled and the corresponding *EnablingFunction()*, and how each event is handled. We list the events as follows, assuming that n_0 is the sender and n_1 is the receiver:

T_0 Delivering a data packet: This event is enabled if the network contains at least one data packet. When enabled, *EnablingFunction(currentState, n_1, T_0)* returns 1. The event is handled by removing this packet from the network and forwarding it to the receiver in order to be processed. If the sequence number in the data packet is equal to $SeqNoExpected$, the receiver updates $SeqNoExpected$ and $NumDistinctDataMsgReceived$. In any case, the receiver transmits an ACK announcing $SeqNoExpected$.

T_1 Delivering an ACK packet: This event is enabled if the network contains at least one ACK packet. When enabled, *EnablingFunction(currentState, n_0, T_1)* returns 1. The event is handled by removing this packet from the network and forwarding it to the sender in order to be processed. If the sequence number in the ACK is equal to $SeqNoSent$, the sender retransmits the data packet with that sequence number; otherwise, the sender updates $SeqNoSent$ and $NumDistinctDataMsgSent$ and transmits a new data packet.

T_2 Timeout of the retransmission timer at the sender: This event is enabled if the network does not contain any data packets. This condition corresponds to either the case that the most recently sent data packet was lost and hence needs to be retransmitted or the case that the ACK of the most recently sent data packet is still in transit from the receiver to the sender. When enabled, *EnablingFunction(currentState, n_0, T_2)* returns 1. The event is handled by having the sender retransmit the data packet that was most recently sent and whose ACK the sender is waiting for.

T_3 Loss of a data packet: This event is enabled if the network contains at least one data packet. When enabled, *EnablingFunction(currentState, n_1, T_3)* returns 1. The event is handled by removing this packet from the network.

T_4 Loss of an ACK packet: This event is enabled if the network contains at least one ACK packet. When enabled, *EnablingFunction(currentState, n_0, T_4)* returns 1. The event is handled by removing this packet from the network.

Step 3. Equality Relation

We use the following equality relation between states. Two states $s_1, s_2 \in S$ are equal if (i) s_1 and s_2 have the same number and the same sequence of identical packets, and (ii) s_1 and s_2 have equal corresponding values for *SeqNoSent*, *SeqNoExpected*, *NumDistinctDataMsgSent*, and *NumDistinctDataMsgReceived*.

Step 4. State Ranking: Exploiting protocol-specific properties

Recall from Chapter 3 that in order to enable a best-first search strategy, the user needs to write a Java method that maps each state to a tuple $< b_1, b_2, \ldots, b_{B-1}, b_B >$ based on protocol-specific properties such that a state s_1 is considered "better than" a state s_2 if s_1 has a higher lexicographical order of this tuple than s_2.

A suitable BeFS heuristic for exploring the state space of ARQ can be obtained by inspecting the assertion, which requires that

$$(NumDistinctDataMsgSent - NumDistinctDataMsgReceived) \leq 2$$

Intuitively, the greater $(NumDistinctDataMsgSent - NumDistinctDataMsgReceived)$ in a state s, the more likely s is close to an error. Therefore, a suitable BeFS heuristic for ARQ is to consider a state s_1 better than another state s_2 if the quantity $(NumDistinctDataMsgSent - NumDistinctDataMsgReceived)$ in s_1 is greater than the corresponding quantity in s_2. In other words, the BeFS tuple, assigned to a state s, consists of one component $< b_1 >$ such that b_1 is the quantity $(NumDistinctDataMsgSent - NumDistinctDataMsgReceived)$ in s. We call this BeFS heuristic ARQ-1-BeFS.

We also consider the following heuristic, which is "badly" obtained from the assertion. Specifically, the BeFS tuple, assigned to a state s, consists of one component $< b_1 >$ such that $b_1 = (NumDistinctDataMsgReceived - NumDistinctDataMsgSent)$, We call this "bad" BeFS heuristic ARQ-2-BeFS.

6.3 Results of the Verification

To demonstrate the ability of the state space exploration framework to find assertion violations, we manually injected the following error in the ACK packet delivery event T_1. Specifically, the sender does not check the sequence number in the ACK before sending a data packet; i.e., the sender always

sends a new data packet whenever an ACK is received. This error may lead to a violation of the assertion because it may make the sender think that a lost/corrupted data packet has been received by the receiver. We call this assertion violation as Counterexample 1.

When running this erroneous simulation model of ARQ in J-Sim, the user may obtain an output trace similar to that shown in Figure 6.1 in a typical J-Sim simulation run. (In Figure 6.1, a data packet with sequence number i is denoted by Di and an ACK with sequence number j is denoted by $ACKj$.) Repeating the same experiment several times may also reproduce the same output. This may lure into believing that this erroneous simulation model of ARQ is correct. However, not making the sender check the sequence number in the ACK before sending a data packet may lead to an error. Using the breadth-first BFS-ANS search strategy, we obtain the counterexample given in Figure 6.2. We explain the counterexample as follows. State 1 is the initial state. In State 2, the receiver receives *D0* and transmits an ACK. In State 3, the sender's retransmission timer expires prematurely causing the sender to retransmit *D0*. In State 4, the receiver discards the duplicate *D0* and retransmits the ACK. In State 5, the sender receives an ACK and transmits *D1*. In State 6, the sender receives the other ACK and since the sender does not check the sequence number in the ACK before sending a data packet, the sender thinks this ACK is acknowledging the receipt of *D1* sent in State 5 and transmits *D0*. In State 7, *D1* is lost; i.e., the receiver will miss *D1* although the sender believes that *D1* has been received by the receiver. In State 8, the receiver discards the duplicate *D0* and retransmits the ACK. In State 9, the sender receives the ACK and since the sender does not check the sequence number in the ACK before sending a data packet, the sender thinks it is acknowledging the receipt of *D0* sent in State 6 and transmits *D1*. State 9 represents a state from which the ARQ protocol can resume without the sender and the receiver noticing that an error has happened. In other words, the ARQ protocol fails.

Table 6.1 gives the performance of the various randomized search strategies with respect to the following two types of performance evaluation criteria: (a) *platform-independent*, namely the number of events executed and the number of states stored in memory (sum of the sizes of *AlreadyVisited* and *ToBeExplored*)[1], and (b) *platform-dependent*, namely the time needed to find an assertion violation. In Table 6.1, *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation. We ran 100 experiments for each search strategy. Each experiment has a different seed, but the same set of 100 seeds were used for each of the seven search strategies. For each performance evaluation criterion, we report the minimum, the maximum

[1]Note that we do not report the total number of states generated because it is simply equal to the number of events executed plus 1 (to account for the initial state).

```
Sender: sending D0
Receiver: receiving EXPECTED D0
Receiver: sending ACK1, expecting D1
Sender: receiving ACK1
Sender: sending D1
Receiver: receiving EXPECTED D1
Receiver: sending ACK0, expecting D0
Sender: receiving ACK0
Sender: sending D0
Receiver: receiving EXPECTED D0
Receiver: sending ACK1, expecting D1
Sender: receiving ACK1
Sender: sending D1
Receiver: receiving EXPECTED D1
Receiver: sending ACK0, expecting D0
Sender: receiving ACK0
Sender: sending D0
Receiver: receiving EXPECTED D0
Receiver: sending ACK1, expecting D1
Sender: receiving ACK1
Sender: sending D1
Receiver: receiving EXPECTED D1
Receiver: sending ACK0, expecting D0
Sender: receiving ACK0
Sender: sending D0
.....................
```

Figure 6.1: A sample output trace of the erroneous simulation model of ARQ. The trace looks like a valid output trace of an error-free simulation model of ARQ. Hence, the user cannot perceive the error.

and the average values. As shown in Table 6.1, best-first search strategies (ARQ-1-BeFS-ACS and ARQ-1-BeFS-ANS) perform better than the corresponding breadth-first search strategies (BFS-ACS and BFS-ANS) in terms of the three performance evaluation criteria. This is because the ARQ-1-BeFS heuristic successfully guides the search towards the states that may potentially lead to a state that violates the assertion. On the other hand, the "bad" ARQ-2-BeFS heuristic failed to guide the search and hence incurred a performance that is significantly worse than both BFS and DFS. This result shows that, even for such a simple case study, the choice of a BeFS heuristic can have a significant effect on the performance of the state space exploration framework.

	Time (ms.)			Space (number of states)			Events		
	Min	Max	Average	Min	Max	Average	Min	Max	Average
DFS-RS	12.0	72.0	41.87	10	50	29.35	11	121	62.77
BFS-ACS	65.0	85.0	73.27	78	104	89.32	109	146	125.34
DFS-ACS	14.0	74.0	44.29	20	63	41.62	22	132	75.13
ARQ-1-BeFS-ACS	61.0	70.0	62.96	50	53	51.39	96	103	99.58
ARQ-2-BeFS-ACS	147.0	154.0	148.37	191	192	191.57	471	475	472.96
BFS-ANS	72.0	86.0	76.38	77	103	87.46	109	146	125.34
DFS-ANS	20.0	75.0	40.91	23	59	40.78	21	120	60.82
ARQ-1-BeFS-ANS	61.0	69.0	62.63	52	55	53.51	96	102	99.06
ARQ-2-BeFS-ANS	117.0	125.0	118.19	147	149	148.00	336	340	338.02

Table 6.1: ARQ case study: Time and space requirements (sum of the sizes of *AlreadyVisited* and *ToBeExplored*) and the number of events executed for finding the counterexample using several randomized search strategies. $MAXDEPTH = 10$. *AlreadyVisited* stores the states that have already been visited and the stateful search depends on the simulation relation.

```
Counterexample
State 1 Depth = 0
        Sender: SeqNoSent=0 NumDistinctDataMsgSent=1
        Receiver: SeqNoExpected=0 NumDistinctDataMsgReceived=0
        Network: D0
State 2 Depth = 1
        Sender: SeqNoSent=0 NumDistinctDataMsgSent=1
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: ACK1
State 3 Depth = 2
        Sender: SeqNoSent=0 NumDistinctDataMsgSent=1
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: D0, ACK1
State 4 Depth = 3
        Sender: SeqNoSent=0 NumDistinctDataMsgSent=1
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: ACK1, ACK1
State 5 Depth = 4
        Sender: SeqNoSent=1 NumDistinctDataMsgSent=2
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: D1, ACK1
State 6 Depth = 5
        Sender: SeqNoSent=0 NumDistinctDataMsgSent=3
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: D1, D0
State 7 Depth = 6
        Sender: SeqNoSent=0 NumDistinctDataMsgSent=3
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: D0
State 8 Depth = 7
        Sender: SeqNoSent=0 NumDistinctDataMsgSent=3
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: ACK1
State 9 Depth = 8
        Sender: SeqNoSent=1 NumDistinctDataMsgSent=4
        Receiver: SeqNoExpected=1 NumDistinctDataMsgReceived=1
        Network: D1
```

Figure 6.2: ARQ case study: Trace of the counterexample obtained using the breadth-first BFS-ANS search strategy.

Chapter 7

Incremental State Space Exploration

In this chapter, we elaborate on the incremental state space exploration technique that we implemented in J-Sim while keeping our two design goals (Section 3.1) met. Note that the incremental state space exploration technique is implemented as a procedure in the state space explorer component (Figure 3.1). In Section 7.1, we review the traditional (non-incremental) state space exploration procedure. In Section 7.2, we present the incremental state space exploration procedure. In Section 7.3, we analyze the performance of each of the non-incremental and incremental state space exploration procedures in order to analytically derive necessary conditions for the incremental state space exploration procedure to provide a speedup in the state space exploration time when compared to the non-incremental state space exploration procedure. Finally, in Section 7.4, we discuss the correctness of both the non-incremental and incremental state space exploration procedures.

7.1 Non-incremental State Space Exploration Procedure

Figure 7.1 shows the pseudo-code of the non-incremental state space exploration procedure SSExplore(), which is similar to SSExploreAddNext() (Figure 3.2) except for the fact that *AlreadyVisited* has to store the *hash codes* of the states that have already been visited instead of the states themselves. Figure 7.1 thus presents an explicit-state stateful search that avoids (re-)exploring a state s_1 if another state s_2, having the same hash code as s_1, has already been visited before. This check is made in Figure 7.1, line 20. In order to enable this check, the hash codes of the visited states have to be computed (Figure 7.1, line 3 and line 13) and added to *AlreadyVisited*, if necessary, to denote that these states have been visited (Figure 7.1, line 4 and line 22). The definitions of the tree and non-tree events (Section 3.3.2) are modified accordingly. For example, a tree event is an event that generates a *nextState* whose depth is strictly less than $MAXDEPTH$ and whose hash code does not exist in *AlreadyVisited* at any depth that is less than or equal to the depth of *nextState*. The definitions of the deepest, safe and unsafe events remain the same as in Section 3.3.2.

```
 1. procedure SSExplore()
 2.    initialState.depth = 0 ;
 3.    initialStateHashCode = initialState.computeHashCode() ;
 4.    AlreadyVisited.add(initialStateHashCode) ;
 5.    ToBeExplored.add(initialState) ;
 6.    while ( ToBeExplored is not empty ) {
 7.       currentState = ToBeExplored.remove() ;
 8.       EnabledEvents = GenerateEnabledEvents(currentState) ; /* See Figure 3.2 for GenerateEnabledEvents() */
 9.       for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
10.          EventInfo E = EnabledEvents.get(i) ;
11.          nextState = GenerateNextState(currentState, E) ; /* X: execute an event */
                                                              /* See Figure 3.2 for GenerateNextState() */
12.          nextState.setDepth(currentState.depth + 1) ;
13.          nextStateHashCode = nextState.computeHashCode() ; /* H: compute a hash code */
14.          checkProperty = nextState.verifyAssertion() ;  /* Y: verify the assertion */
15.          if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
16.             Print("Counterexample ") ;
17.             printCounterexample(nextState) ;
18.             exit ;
19.          } else if ( (checkProperty == true) AND (nextState.depth < MAXDEPTH) ) {
                /* v: safe event; i.e., nextState does not violate any assertion. 1-v: unsafe event. */
                /* d: deepest event; i.e., depth of nextState == MAXDEPTH. */
20.             if ( nextStateHashCode does not exist in AlreadyVisited ) { /* A: search in AlreadyVisited */
                   /* n: non-tree event. t=1-n-d: tree event. */
21.                if ( search strategy is best-first ) nextState.computeBeFSTuple() ;  /* U: compute BeFS tuple */
22.                AlreadyVisited.add(nextStateHashCode) ;  /* R: add a hash code to AlreadyVisited */
23.                ToBeExplored.add(nextState) ;  /* N: add a state to ToBeExplored */
                }
             }
          }
       }
```

Figure 7.1: An explicit-state stateful state space exploration procedure. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes. The symbols used in the comments are explained in Table 7.2.

Similar to SSExploreAddNext() (Figure 3.2), SSExplore() (Figure 7.1) invokes *GenerateEnabledEvents()* (Figure 3.2, lines 23-30) and *GenerateNextState()* (Figure 3.2, lines 31-36) to determine the enabled events and execute them respectively. Furthermore, SSExplore() can employ the different (randomized) search strategies: BFS-AN, DFS-AN, BeFS-AN, BFS-ANS, DFS-ANS and BeFS-ANS. The best-first search strategies are implemented by state ranking (Figure 7.1, line 21). The randomized search strategies are implemented by shuffling the set of enabled events at each state being explored as explained in Section 3.3.4.

7.2 Incremental State Space Exploration Procedure

Incremental state space exploration (ISSE) is a technique that aims to provide a speedup in the verification time of evolving simulation models. Incremental state space exploration considers *several versions* of the simulation model and reuses the results of verifying one version to speed up the verification of a subsequent one. We first mention the assumptions of ISSE and then describe the technique.

7.2.1 Assumptions

We make the following two assumptions:

1. The first assumption is related to the definition of the global state (Section 3.2.1): The definition of the global state does not change from one version to another; e.g., a new version does not add or remove attributes to the existing global state definition. In particular, the implementation of the function $computeHashCode()$ that computes the hash code of a state as part of the verification model does not change in the subsequent version(s) verified using ISSE.

2. The second assumption is related to the event IDs and the event parameters (Section 3.3.2): An event having ID e in one version keeps the same ID in the subsequent version(s) verified using ISSE. In addition, the meanings of the event parameters stay the same from one version to another. We elaborate on this assumption in the following section.

7.2.2 Technique

We describe the ISSE technique in the context where the evolution from one version of the simulation model to another is the modification of one or more of the existing event handlers in the simulation model. In order to use the ISSE technique, the extra work that the user needs to do is to determine which events were modified. The ISSE technique is also applicable, in a straightforward way, to other kinds of evolutions such as adding a new event handler and/or removing an existing one as long as the second assumption mentioned in Section 7.2.1 holds. In the case of adding a new event, the new event will have a new event ID that is not used by any old event. In the case of both adding a new event and removing an existing one, the event ID assigned to the new event has to be unequal to the ID that was used by the event being removed. (In some of our experiments (Chapter 8), we studied the effect of adding a new event handler.)

Figure 7.3 shows the pseudo-code of IncrementalSSExplore(), the incremental version of the state space exploration procedure in Figure 7.1. The added or modified lines are shown in italic. In Figure 7.3, line 1, the two parameters *write* and *read* determine the mode of operation of the procedure. Table 7.1 shows the four possible modes of operation of the ISSE technique.

If *write* is $false$ and *read* is $false$, it is easy to see that the operation of IncrementalSSExplore() is exactly the same as SSExplore() (Figure 7.1). In other words, the non-incremental SSE procedure is a special case of the ISSE technique.

read	*write*	Mode
false	false	no read, no write (non-incremental SSE)
false	true	write-only
true	false	read-only
true	true	read-write

Table 7.1: The four possible modes of operation of the ISSE technique. The non-incremental SSE procedure (Section 7.1) is just one mode of these four modes.

If *write* is *true*, IncrementalSSExplore() stores the state space graph in the $OUTPUT$ data structure (Figure 7.3, lines 42-45). $OUTPUT$ stores only the hash codes of the visited states that do *not* violate an assertion. Specifically, each executed safe event $s \xrightarrow{E} s'$, where E is an instance of *EventInfo* (Section 3.3.2) and both s and s' are instances of *GlobalState* (Section 3.2.1), is represented in $OUTPUT$ as $< h, E, h' >$ where h is the hash code of s and h' is the hash code of s'. Before IncrementalSSExplore() terminates, the $OUTPUT$ data structure is written to an output file (Figure 7.3, line 35 and line 46) on secondary storage.

If *read* is *true*, IncrementalSSExplore() first loads a state space graph from an input file and stores it in the $INPUT$ data structure (Figure 7.3, line 2). In a typical setting, the user would invoke IncrementalSSExplore() with $read = false$ and $write = true$ in order to write the state space graph of one version to an output file. Following that, while verifying a subsequent version, the user would invoke IncrementalSSExplore() with $read = true$ in order to load the state space graph of the previous version from that file.

For each enabled event in *currentState*, IncrementalSSExplore() makes use of two decision variables *execute* and *postProcess* to determine the "workload" associated with this enabled event. If *execute* is $false$, this enabled event is not executed and hence no workload is associated with it. If *execute* is *true* (Figure 7.3, line 23), the event is executed and a *nextState* is generated (Figure 7.3, line 24). If *postProcess* is also *true* (Figure 7.3, line 26 and line 38), (i) the hash code of *nextState* has to be computed (Figure 7.3, line 27), (ii) IncrementalSSExplore() has to check whether *nextState* violates an assertion (Figure 7.3, line 28), and (iii) IncrementalSSExplore() may have to check whether the hash code of *nextState* exists in *AlreadyVisited* (Figure 7.3, line 38). Otherwise, none of these three operations need to be done.

The settings of *execute* and *postProcess* are determined as follows. Initially, both are set to *true* (Figure 7.3, lines 12-13). IncrementalSSExplore() then checks whether the enabled event represents a non-modified event. This check is achieved by the *isModifiedEvent()* function call (Figure 7.3, line 15), which returns true if the event is modified and false otherwise. As mentioned above, the

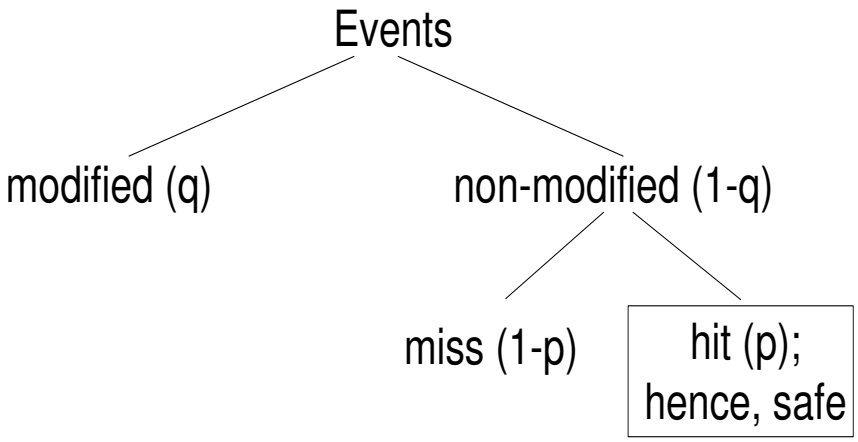


Figure 7.2: Non-modified events can be either "miss" or "hit". Potential savings obtained from ISSE (indicated by the rectangle) come from the non-modified hit events. The symbols used between parentheses are explained in Table 7.2.

implementation of *isModifiedEvent()* is the responsibility of the user.

If the event is modified (i.e., *isModifiedEvent()* returns true), *execute* and *postProcess* remain *true* since the event has to be executed and the generated *nextState* has to be post-processed as explained above. On the other hand, if the enabled event represents a non-modified event, potential savings may be gained depending on whether or not information about this event exists in $INPUT$. IncrementalSSExplore() searches for the hash code of *currentState* and the *EventInfo* E that corresponds to this event in $INPUT$ (Figure 7.3, line 16). We here divide the non-modified events into two disjoint types:

1. *miss event*: a non-modified event such that the hash code of *currentState* and the *EventInfo* E that corresponds to this event do not exist in $INPUT$.

2. *hit event*: a non-modified event such that the hash code of *currentState* and the *EventInfo* E that corresponds to this event exist in $INPUT$.

Figure 7.2 shows this division diagrammatically.

If the hash code of *currentState* and the *EventInfo* E that corresponds to the non-modified event do not exist in $INPUT$ (i.e., the case of a miss event), then *execute* and *postProcess* remain *true* since this event may generate a *nextState* that violates an assertion (i.e., the case of an unsafe event) or a *nextState* that has not been visited before (i.e., the case of a safe tree event). On the other hand, if the hash code of *currentState* and the *EventInfo* E that corresponds to the non-modified event exist in $INPUT$ (i.e., the case of a hit event), then *postProcess* is set to $false$ (Figure 7.3,

line 17) because we can fetch the hash code of *nextState* from $INPUT$ (Figure 7.3, line 18) and we know that *nextState* does not violate an assertion because $INPUT$ does not store the hash codes of states that violate an assertion as explained above; i.e., every hit event is safe. Hence, we saved both the time of computing the hash code of *nextState* and the time of checking whether *nextState* violates an assertion.

Following that, IncrementalSSExplore() checks if the depth of *nextState* (which is one plus the depth of *currentState*) is equal to $MAXDEPTH$. If so (i.e., the case of a hit deepest event), *execute* is also set to $false$ (Figure 7.3, line 20) because there is no need to execute an event that will generate a *nextState* that satisfies the assertion and will not be added to $ToBeExplored$. If not, IncrementalSSExplore() checks if the hash code of *nextState* exists in *AlreadyVisited* (Figure 7.3, line 21). If so (i.e., the case of a hit non-tree event), *execute* is also set to $false$ (Figure 7.3, line 22) for the same reason as in the former check. If not (i.e., the case of a hit tree event), *execute* remains $true$.

In summary, the potential savings obtained from the ISSE technique come from two primary sources:

1. non-modified hit tree events: not computing the hash code of *nextState* and not checking whether *nextState* violates an assertion, and

2. non-modified hit non-tree or deepest events: not executing the event, not computing the hash code of *nextState* and not checking whether *nextState* violates an assertion.

It is important to note that every tree event *is* executed.

It should be mentioned that if both $read = true$ and $write = true$, we use only one data structure for both $INPUT$ and $OUTPUT$ instead of two separate data structures. This design choice saves the time of inserting in $OUTPUT$ the event information that already exists in $INPUT$. This explains why the conditions in the if statements (Figure 7.3, lines 42-44) include the condition $postProcess == true$.

Note that an obvious overhead that is incurred by the incremental state space exploration technique is the extra memory overhead due to the $INPUT$ and/or $OUTPUT$ data structures. Note further that while loading a state space graph from an input file and storing it in the $INPUT$ data structure (Figure 7.3, line 2), we do *not* insert the information of the modified events in $INPUT$. This reduces the memory consumed by $INPUT$ and at the same time does not lose any potential time savings since we search in $INPUT$ (Figure 7.3, line 16) for the non-modified events only

(Figure 7.3, line 15).

In the following chapter, we measure the memory overhead incurred by the incremental state space exploration technique in our experiments. In the next section, we focus on the performance of the incremental state space exploration technique with respect to time.

7.3 Analysis

In this section, we analyze the performance of each of the non-incremental and incremental state space exploration procedures in order to analytically derive necessary conditions for the incremental state space exploration procedure to provide a speedup in the state space exploration time when compared to the non-incremental state space exploration procedure.

7.3.1 Analysis of the Non-incremental State Space Exploration Procedure

First, we analyze the average time T_{NonInc} spent in an iteration of the for loop in which enabled events get executed and the generated states are post-processed (Figure 7.1, lines 9-23). We focus on the operations that would help us compare between the non-incremental and the incremental state space exploration procedures. Table 7.2 shows the notations that we use in this chapter. Some of the comments in Figure 7.1 make use of the symbols shown in Table 7.2.

Analyzing Figure 7.1, we see that

$$T_{NonInc} = X + H + Y + vnA + vt(A + U + R + N)$$

Note that $n + t = 1 - d$. Hence, we have,

$$T_{NonInc} = X + H + Y + v(1 - d)A + vt(U + R + N)$$

7.3.2 Analysis of the Incremental State Space Exploration Procedure

Next, we analyze the average time T_{Inc} spent in an iteration of the for loop in which enabled events may or may not be executed and the generated states may or may not be post-processed (Figure 7.3, lines 10-45). Again, we focus on the operations that would help us compare between the non-incremental and the incremental state space exploration procedures. The purpose of this analysis is to identify the necessary conditions for the incremental state space exploration procedure to provide a speedup in the state space exploration time.

X	Time to execute an enabled event.
H	Time to compute the hash code of a generated state.
Y	Time to check whether or not a generated state satisfies the assertion.
v	Proportion of explored events that are safe. Hence, $1 - v$ is the proportion of explored events that are unsafe.
d	Proportion of explored events that are deepest.
n	Proportion of explored events that are non-tree.
t	Proportion of explored events that are tree. $t = 1 - n - d$
A	Time to search in *AlreadyVisited*.
U	Time to compute the BeFS tuple of a generated state. $U = 0$ if the search strategy is not best-first.
R	Time to add a hash code of a generated state to *AlreadyVisited*.
N	Time to add a generated state to *ToBeExplored*.
q	Proportion of explored events that are modified. Hence, $1 - q$ is the proportion of explored events that are not modified.
L	Time to search in $INPUT$.
p	Proportion of explored non-modified events whose information is found in $INPUT$. In other words, p is the proportion of explored non-modified events that are hit (i.e., searches in $INPUT$ that are successful). Hence, $1 - p$ is the proportion of explored non-modified events that are miss.
F_{total}	Time to load a state space graph from the input file and store it in $INPUT$.
κ	Total number of explored events.
F	$\frac{F_{total}}{\kappa}$

Table 7.2: Notation used in this chapter: A capital letter is used for the average time spent in an operation, and a small letter is used for the proportion of events that satisfy a certain condition.

First we note that an obvious overhead of IncrementalSSExplore() is the time F_{total} spent in loading the state space graph from the input file and storing it in $INPUT$ (Figure 7.3, line 2). Since we are interested in analyzing the average time spent in an iteration of the for loop (Figure 7.3, lines 10-45), we work with $F = \frac{F_{total}}{\kappa}$ where κ is the total number of explored events.

Obviously, if $read$ is $false$, the incremental state space exploration technique is disabled, and hence no speedup is expected. Furthermore, if $q = 1$ (i.e., all enabled events are modified) then the condition in the if statement (Figure 7.3, line 15) will always be false, and no speedup will be expected from the incremental state space exploration procedure. Hence, if the user determines that all events were modified, we set $read$ to $false$ in order to save the time that would otherwise be wasted in opening and processing the input file (Figure 7.3, line 2).

Similarly, if $p = 0$ (i.e., no information about the explored non-modified events is found in $INPUT$) then the condition in the if statement (Figure 7.3, line 16) will always be false, and no speedup will be expected from the incremental state space exploration procedure either. Therefore, $read = true$, $q \neq 1$ and $p \neq 0$ are three obvious necessary conditions. Another trivial necessary condition is $v \neq 0$.

Analyzing Figure 7.3, we see that the average time T_{Inc} spent in an iteration of the for loop (Figure 7.3, lines 10-45) is given by:

$$T_{Inc} = F + qT_{NonInc} + (1-q)\{L + (1-p)T_{NonInc} + pnA + pt[A + X + U + R + N]\}$$

In order to have a speedup, we must have $T_{Inc} < T_{NonInc}$. In other words,

$$F + qT_{NonInc} + (1-q)\{L + (1-p)T_{NonInc} + p(1-d)A + pt(X + U + R + N)\} < T_{NonInc}$$

If $q \neq 1$, we have

$$\frac{F}{1-q} + L + (1-p)T_{NonInc} + p(1-d)A + ptX + pt(U + R + N) < T_{NonInc}$$

If $p \neq 0$, we have

$$\frac{F}{p(1-q)} + \frac{L}{p} + (1-d)A + tX + t(U + R + N) < T_{NonInc}$$

Recall that $T_{NonInc} = X + H + Y + v(1-d)A + vt(U + R + N)$. Hence, we have

$$\frac{F}{p(1-q)} + \frac{L}{p} + (1-d)A + tX + t(U + R + N) < X + H + Y + v(1-d)A + vt(U + R + N)$$

$$\frac{F}{p(1-q)} + \frac{L}{p} + (1-d)(1-v)A + t(1-v)(U + R + N) < H + Y + (1-t)X$$

Since in the state spaces of the sophisticated simulation models of network protocols, the proportion of explored events that are unsafe is usually significantly small, it is reasonable to assume that $v \approx 1$. This is especially true if the state space explorer searches for a state that violates the assertion and terminates the state space exploration once one is found. Therefore, and since we are only interested in a necessary condition and the two terms $(1-d)(1-v)A$ and $t(1-v)(U+R+N)$ are non-negative, if both A and $(U + R + N)$ are of the same order[1] as $\frac{F}{p(1-q)} + \frac{L}{p}$, we can drop the two terms $(1-d)(1-v)A$ and $t(1-v)(U+R+N)$ and get the following (simplified) necessary condition:

$$\frac{F}{p(1-q)} + \frac{L}{p} < H + Y + (1-t)X$$

Informally, this condition means that incremental state space exploration cannot provide a speedup unless the costs of executing events, computing hash codes, and checking the assertions are high and there is a small proportion of tree events. Furthermore, this condition also says that the larger the value of p and the smaller the value of q, the more likely it is for the incremental state space exploration to provide a speedup. In addition, this condition also suggests that a reduction in p (with q fixed) will have a more adversarial effect on the incremental state space exploration than

[1] We will empirically validate this assumption in the evaluation sections (Chapter 8).

an increase in q (with p fixed). Intuitively, this is true because the overhead associated with a modified event is executing the event and post-processing *nextState* while the overhead associated with a non-modified event whose event information is not found in $INPUT$ is executing the event, post-processing *nextState*, *and* the search in $INPUT$ (Figure 7.3, line 16).

In summary, the necessary conditions for IncrementalSSExplore() (Figure 7.3) to provide a speedup in state space exploration when compared with SSExplore() (Figure 7.1) are:

1. $read = true$
2. $q \neq 1$, $p \neq 0$, and $v \neq 0$
3. $\frac{F}{p(1-q)} + \frac{L}{p} < H + Y + (1-t)X$

The J-Sim state space explorer dynamically estimates the values of the symbols in Table 7.2 and checks whether or not the necessary conditions are satisfied. The user may run a sample run of IncrementalSSExplore() to know whether using the incremental state space exploration procedure can provide a speedup.

The analysis in this section assumed $write = false$. The case of $write = true$ can be done in a similar way to what we did above.

7.4 Correctness

In this section, we discuss the correctness of both the non-incremental and incremental state space exploration procedures.

First, we consider the correctness of a special case of the procedures, namely when there is no hashing; i.e., the *computeHashCode()* function (Figure 7.1, line 3 and line 13; Figure 7.3, line 4 and line 27) is the identity function. In this case, the non-incremental state space exploration procedure (Figure 7.1) is guaranteed to discover any state that violates an assertion and whose depth is at most $MAXDEPTH$. The incremental state space exploration procedure (Figure 7.3) can also be easily shown to find any such state.

In the presence of hashing, however, there are a couple of subtle issues to consider. First, the non-incremental state space exploration procedure (Figure 7.1) becomes only sound but *not* complete. In other words, it can miss a state that violates an assertion and whose depth is at most $MAXDEPTH$. This is because of the following possible scenario: the procedure may visit a state s_2 that is distinct from a previously visited state s_1 but whose hash code is equal to the hash code

of s_1. Hence, when the non-incremental state space exploration procedure looks up the hash code of s_2 in *AlreadyVisited* (Figure 7.1, line 20), it will decide to not explore further any executions from s_2. This can potentially miss those states that are reachable from s_2 and violate an assertion. Since the non-incremental state space exploration procedure is not complete in the presence of hashing, its incremental version (Figure 7.3) will clearly also not be complete due to the same reasons.

However, there is a second subtle source for the incompleteness of the incremental state space exploration procedure in the presence of hashing. Suppose that, while verifying a previous version of the simulation model, a state s_1 is explored and its hash code and events information are output to the $OUTPUT$ data structure and eventually to the output file on secondary storage. Suppose further that, while verifying a subsequent version of the model after loading the state space graph from that file into the $INPUT$ data structure, a state s_2 is visited before s_1 such that (1) s_2 is distinct from s_1 but the hash code of s_2 is equal to that of s_1, and (2) s_2 has an enabled non-modified event that was also enabled in s_1. In this case, the incremental state space exploration procedure will (incorrectly) conclude that the state s_2 was visited while verifying the previous version of the model (Figure 7.3, line 16). The incremental state space exploration procedure may then use the hash code(s) of the state(s) reached from s_1 (Figure 7.3, line 18) to search in *AlreadyVisited* (Figure 7.3, line 21). Consequently, the incremental procedure may not explore certain paths that would be explored by the non-incremental version.

Despite these limitations, the incremental procedure is *guaranteed to be sound* for assertions, which are the focus of this thesis. This is because every *tree* event is executed, even if the hash code of the generated *nextState* is not recomputed. Since the discovery of assertion violations is often the goal of state space exploration tools (as witnessed by the widespread use of hashing in non-incremental procedures despite its incompleteness), we expect that these theoretical limitations will not affect the usefulness of our techniques in practice.

```
 1. procedure IncrementalSSExplore(boolean write, boolean read)
 2. if ( read ) INPUT = readFromInputFile() ;
 3. initialState.depth = 0 ;
 4. initialStateHashCode = initialState.computeHashCode() ;
 5. AlreadyVisited.add(initialStateHashCode) ;
 6. ToBeExplored.add(initialState) ;
 7. while ( ToBeExplored is not empty ) {
 8.    currentState = ToBeExplored.remove() ;
 9.    EnabledEvents = GenerateEnabledEvents(currentState) ;  /* See Figure 3.2 for GenerateEnabledEvents() */
10.    for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
11.       EventInfo E = EnabledEvents.get(i) ;
12.       execute = true ;
13.       postProcess = true ;
14.       if ( read ) {
15.          if ( isModifiedEvent(E.getEventID()) == false ) {
                /* q: event is modified. 1-q: event is not modified. */
16.             if ( <currentState.hashCode, E> exists in INPUT ) { /* L: search in INPUT */
                   /* p: event information is found in INPUT. 1-p: not found in INPUT. */
17.                postProcess = false ;
18.                nextStateHashCode = <currentState.hashCode, E>.getNextStateHashCode() ;
19.                if ( (currentState.depth + 1) == MAXDEPTH ) {
                      /* d: deepest event; i.e., depth of nextState == MAXDEPTH. */
20.                   execute = false ;
21.                } else if ( nextStateHashCode exists in AlreadyVisited ) { /* A: search in AlreadyVisited */
                         /* n: non-tree event. t=1-n-d: tree event. */
22.                      execute = false ;
                   }
                }
             }
          }
23.       if ( execute ) {
24.          nextState = GenerateNextState(currentState, E) ; /* X: execute an event */
                                                              /* See Figure 3.2 for GenerateNextState() */
25.          nextState.setDepth(currentState.depth + 1) ;
26.          if ( postProcess ) {
27.             nextStateHashCode = nextState.computeHashCode() ; /* H: compute a hash code */
28.             checkProperty = nextState.verifyAssertion() ; /* Y: verify the assertion */
29.          } else {
30.             nextState.setHashCode(nextStateHashCode) ; /* Hash code of nextState was obtained from INPUT. */
31.             checkProperty = true ; /* INPUT does not store hash codes of states that violate an assertion */
             }
32.          if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
33.             Print("Counterexample ") ;
34.             printCounterexample(nextState) ;
35.             if ( write ) writeToOutputFile(OUTPUT) ;
36.             exit ;
37.          } else if ( (checkProperty == true) AND (nextState.depth < MAXDEPTH) ) {
                /* v: safe event; i.e., nextState does not violate any assertion. */
                /* d: deepest event; i.e., depth of nextState == MAXDEPTH. */
38.             if ( (postProcess == false) OR (nextStateHashCode does not exist in AlreadyVisited ) {
                   /* A: search in AlreadyVisited */
                   /* n: non-tree event. t=1-n-d: tree event. */
39.                if ( search strategy is best-first ) nextState.computeBeFSTuple() ; /* U: compute BeFS tuple */
40.                AlreadyVisited.add(nextStateHashCode) ; /* R: add a hash code to AlreadyVisited */
41.                ToBeExplored.add(nextState) ; /* N: add a state to ToBeExplored */
42.                if ( write AND postProcess ) OUTPUT.add(<currentState.hashCode, E, nextStateHashCode>) ;
43.             } else if ( write AND postProcess ) OUTPUT.add(<currentState.hashCode, E, nextStateHashCode>) ;
44.          } else if ( checkProperty AND write AND postProcess )
45.                   OUTPUT.add(<currentState.hashCode, E, nextStateHashCode>) ;
          }
       }
    }
46. if ( write ) writeToOutputFile(OUTPUT) ;
```

Figure 7.3: An incremental version of the state space exploration procedure in Figure 7.1. Added or modified lines are shown in italic. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes. The symbols used in the comments are explained in Table 7.2.

Chapter 8

Incremental State Space Exploration Experiments

In this chapter, we apply the incremental state space exploration (ISSE) technique to the three case studies: (a) the AODV case study (Section 8.1), (b) the Directed Diffusion case study (Section 8.2), and (c) the ARQ case study (Section 8.3) and compare its performance to that of the traditional, non-incremental state space exploration. We study scenarios in which code changes may or may not lead to behavioral changes. In each experiment, we have checked that the state spaces explored by the non-incremental and the incremental state space explorations are the same. In other words, both techniques have the same "workload". This check is important because it shows that the source of incompleteness due to ISSE, which we discussed in Section 7.4, did not appear in any of our experiments; hence, the time speedup gained by ISSE is *not* because ISSE is exploring a smaller state space than that explored by non-incremental SSE.

In each case study, we used the same definitions of the global state, the initial state, the assertion, and the events mentioned in the corresponding case study in Chapters 4-6. The hash code of a state is computed by first constructing an integer array representation of the state. Following that, an integer-valued hash code of this integer array representation is computed using the Jenkins' hash function that we borrowed from JPF. (We slightly modified this function to return an integer instead of a long value.)

8.1 AODV Case Study (Revisited)

We apply the non-incremental and the incremental state space exploration procedures to the J-Sim simulation model of the Ad-hoc On-Demand Distance Vector (AODV) routing protocol [78] for wireless ad-hoc networks (Section 4.1).

The integer array representation of a state depends on the protocol-specific information such as the AODV packet headers and payloads, and each node's $seqno_n$, bid_n, routing table entries and broadcast ID cache entries. The BeFS heuristic that we consider in the experiments is AODV-3-

BeFS.

We conduct the experiments in this section on a 4 CPU Intel Pentium III 500 MHz machine running Linux version 2.6.9 with 1 GB memory. We use Sun's 1.6.0 b105 Java HotSpot^{TM} Client VM with 0.5 GB initial heap size and 2.5 GB maximum heap size.

8.1.1 Experiments

Recall that while verifying the J-Sim simulation model of AODV (Section 4.3), we manually injected two errors (which we called Counterexamples 2 and 3 respectively): in Counterexample 2, $seqno_{n,d}$ is not incremented when a RTE is invalidated and in Counterexample 3, a RTE is deleted (instead of invalidated) when its lifetime expires. In *all* the experiments of the AODV case study in this section, we require that the counterexample contain at least one state that is generated due to the route timeout event, T_3. In order to achieve that, we made use of the *DoesCounterexampleContainEvent()* function provided by the state space exploration framework (Figure 7.1, line 15 and Figure 7.3, line 32). This effectively prevents the state space explorer from outputting Counterexample 1 mentioned in Section 4.3. The reason why we did that in this section is because we want to focus on studying the evolution of the simulation model from versions that output a counterexample to a version that does not output any counterexample.

Code changes that do not incur behavioral changes:

We first study the performance of the incremental state space exploration procedure in six different scenarios: (i) a best-case scenario ($q = 0$): no events are modified, and (ii) five practical-case scenarios ($0 < q < 1$): one event (the route timeout event T_3) or two events (T_3 and T_4), or three events (T_0, T_3 and T_4), or four events (T_0, T_2, T_3 and T_4), or five events (T_0, T_2, T_3, T_4 and T_5) are modified. Furthermore, we compare between the non-incremental and the incremental state space exploration procedures in each of these six scenarios using the three randomized search strategies BFS-ANS, DFS-ANS, and AODV-3-BeFS-ANS. To isolate the savings gained (or overhead incurred) by the incremental state space exploration procedure, the modified events, if any, are just "flagged" as modified but their implementations are not changed. Practically, this corresponds to code changes that do *not* incur any behavioral changes; e.g., a code refactoring where any changes to the code improve its readability or simplify its structure but do not change its results.

Table 8.1 gives the time needed to find an assertion violation if any exists (Counterexample 2 or 3). In Table 8.1, "Non-inc. SSE" refers to Figure 7.3 with $read = false$ and $write = false$, while

"Incremental SSE" refers to Figure 7.3 with $read = true$ and $write = false$. Before running the "Incremental SSE", we had to execute Figure 7.3 with $read = false$ and $write = true$ in order to generate the file that contains the state space graph; however, the time needed for this intermediate step is not reported. For each experiment, we ran 10 replications. Each replication has a different seed. The savings gained (or overhead incurred) by the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. In this chapter, we denote a saving by a positive sign and an overhead by a negative sign. Formally, the time savings or overhead is given by $\frac{SSE\ time\ -\ ISSE\ time}{SSE\ time}$. The last column of Table 8.1 shows t, which is the proportion of explored events that are tree.

As shown in Table 8.1, the incremental state space exploration technique can indeed provide time savings in the best-case scenario ($q = 0$) and all the practical-case scenarios ($0 < q < 1$). In each row for the incremental state space exploration experiments, the time savings decrease with the increase in q as expected. Note that we implemented our own file read and write functions because we noticed an unreasonably large overhead when we used Java serialization.

In order to understand why the incremental state space exploration procedure was successful in providing time savings, we show in Table 8.2 a breakdown of the average state space exploration time spent in some selected operations taking the first row in Table 8.1 as an example. As shown in Table 8.2, the costs of executing events (i.e., the sum of copying from the verification model to the simulation model, executing the event handlers in the simulation model, and copying from the simulation model to the verification model), computing hash codes, and checking the assertion are considerably high taking together more than 78% of the average total time in the non-incremental state space exploration. In contrast, the times spent in these three operations in the incremental state space exploration procedure ($0 \leq q < 1$) are smaller than their counterparts in the non-incremental exploration. Furthermore, for all the values of q shown in Table 8.2, other operations that are only done in the incremental state space exploration procedure (e.g., reading from the input file and searching in $INPUT$) take a comparatively small amount of time (less than 13% of the incremental state space exploration time). Note that in this particular example, $t = 6.98\%$; i.e., the proportion of events that are tree is small. Furthermore, $p = 0.999$; i.e., almost all of the searches in $INPUT$ are successful. All these observations explain why the incremental state space exploration procedure ($0 \leq q < 1$) provided a time speedup. The second-to-last row in Table 8.2 shows the value of this speedup, which is defined as the non-incremental state space exploration time divided by the incremental state space exploration time; i.e., $\frac{SSE\ time}{ISSE\ time}$. The last row in Table 8.2 gives

No Counterexample $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time Savings (+) or Overhead (-) shown as a percentage of Non-inc. SSE Time: No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events	t
BFS-ANS	125.344	74.11%	73.60%	50.07%	40.42%	33.36%	14.75%	0.0698
DFS-ANS	127.905	76.26%	75.33%	51.40%	41.14%	34.20%	13.24%	0.0704
AODV-3-BeFS-ANS	129.359	75.76%	75.11%	50.46%	40.37%	33.37%	13.77%	0.0706

Counterexample 2 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time Savings (+) or Overhead (-) shown as a percentage of Non-inc. SSE Time: No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events	t
BFS-ANS	63.444	62.12%	61.38%	42.55%	33.54%	28.31%	12.27%	0.1635
DFS-ANS	35.912	74.67%	73.41%	50.10%	39.92%	33.23%	13.86%	0.0766
AODV-3-BeFS-ANS	5.078	65.32%	63.42%	43.78%	33.19%	26.02%	10.92%	0.2094

Counterexample 3 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time Savings (+) or Overhead (-) shown as a percentage of Non-inc. SSE Time: No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events	t
BFS-ANS	60.141	60.66%	60.18%	41.32%	32.27%	26.43%	9.87%	0.1687
DFS-ANS	36.188	74.11%	73.46%	50.14%	39.17%	32.17%	12.93%	0.0755
AODV-3-BeFS-ANS	5.076	65.66%	64.24%	44.54%	34.24%	27.22%	10.35%	0.2094

Counterexample 3 Search Strategy is AODV-3-BeFS-ANS		Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time Savings (+) or Overhead (-) shown as a percentage of Non-inc. SSE Time: No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events	t
N	$MAXDEPTH$								
3	15	0.568	14.20%	12.38%	9.25%	7.56%	5.14%	2.43%	0.5492
4	20	1.521	25.16%	18.68%	12.81%	11.56%	7.17%	3.51%	0.4832
5	25	14.046	49.15%	41.96%	31.34%	30.81%	20.24%	10.99%	0.3991

Table 8.1: AODV case study: Average time (sec.) for non-incremental state space exploration. The savings (+) or overhead (-) of the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. The ad-hoc network consists of N nodes arranged in a chain network topology. The state space explorer terminates state space exploration if an assertion violation is detected. $p = 0.999$. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

the number of the executed events. Note that although $t = 6.98\%$, the time of executing the event handlers in the incremental state space exploration with no modified events is larger than 6.98% of the time of executing the event handlers in the non-incremental state space exploration. We have found that the reason for this observation is that the average time of executing a tree event is more than the average time of executing a non-tree or deepest event.

It should also be noticed that the times of the operations that do not appear in the simplified necessary condition (e.g., inserting the hash code of a state in *AlreadyVisited*, inserting a state in *ToBeExplored*, searching in *AlreadyVisited*, and generating the enabled events) are almost equal in both the non-incremental and incremental state space exploration techniques. Furthermore, the sum of the times to search in *AlreadyVisited*, insert a hash code in *AlreadyVisited*, and insert a state in *ToBeExplored* is of the same order as the sum of the times to search in $INPUT$ and read from the input file in the incremental state space exploration technique for all the values of q shown in Table 8.2. This justifies the assumption that we made in Section 7.3.2.

No Counterexample $N = 3$ $MAXDEPTH = 9$ BFS-ANS Search Strategy Operation	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time (sec.)					
		No Modified Events q=0	One Modified Event q=0.0108	Two Modified Events q=0.2856	Three Modified Events q=0.4143	Four Modified Events q=0.5130	Five Modified Events q=0.7907
From V model to S model	14.012	1.303	1.613	5.102	6.543	7.793	11.236
Execute event handlers	46.980	5.871	6.155	20.235	25.667	29.397	38.703
From S model to V model	13.083	1.088	1.277	4.784	6.512	7.763	10.839
Compute a hash code	16.311	0.097	0.297	5.100	7.253	8.963	13.036
Verify an assertion	7.554	0.001	0.113	2.218	3.126	3.877	5.907
Read from input file	0.000	0.555	0.555	0.505	0.487	0.470	0.422
Search in $INPUT$	0.000	3.562	3.458	2.647	2.204	1.867	0.997
Insert in $AlreadyVisited$	0.901	1.206	1.118	1.046	0.991	1.034	0.989
Search in $AlreadyVisited$	1.074	0.947	0.874	0.935	0.993	1.029	1.015
Generate enabled events	4.628	4.943	4.816	5.131	4.545	4.486	4.498
Insert in $ToBeExplored$	3.519	4.002	3.923	3.810	3.887	3.628	3.523
Compute the BeFS tuple	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Subtotal	108.062	23.575	24.199	51.513	62.208	70.307	91.165
Other operations	17.282	8.873	8.893	11.069	12.467	13.221	15.686
Total time	125.344	32.448	33.092	62.582	74.675	83.528	106.851
Speedup		3.86x	3.79x	2.00x	1.68x	1.50x	1.17x
Number of executed events	460150	32114	36228	151658	206985	247422	369781

Table 8.2: AODV case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations. The example shown corresponds to the first row in Table 8.1. $p = 0.999$

As mentioned in Section 7.2.2, an obvious overhead that is incurred by the incremental state space exploration technique is the extra memory overhead due to the $INPUT$ and/or $OUTPUT$ data structures. In Table 8.3, we measure this memory overhead for the experiments shown in Table 8.1 and report it as a percentage of the corresponding average non-incremental state space exploration memory consumption. Formally, the memory overhead is given by $\frac{SSE\ memory\ -\ ISSE\ memory}{SSE\ memory}$. In this chapter, we measure the memory consumption of each technique by subtracting the total amount of memory currently available for future allocated objects (returned by Runtime.freeMemory()) from the total amount of memory currently available for current and future objects (returned by Runtime.totalMemory()). Note that this measurement does *not* give an accurate value for the memory consumed in each technique because it only measures the object heap, which is not the only factor that contributes to a program's memory footprint; other factors being classes, native data structures and code, etc. In each replication, this measurement is repeated periodically every 500 transitions and also done at the end of the replication, after running garbage collection each time, and we keep track of the maximum memory consumption among the samples collected during the replication. Note that this sampling mechanism does *not* accurately give the maximum memory required for state space exploration because it only measures the maximum memory consumption among the samples collected. Despite these limitations, our measurements do serve our purpose since we are only interested in the relative *difference* between the non-incremental and incremental state space exploration techniques due to the incremental state space exploration memory overhead, and since the state spaces created by the non-incremental and the incremental state space explorations are

No Counterexample $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Memory Overhead (-) shown as a percentage of Non-inc. SSE Memory					
		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events
BFS-ANS	75.569	-15.10%	-15.00%	-12.45%	-11.26%	-10.34%	-7.76%
DFS-ANS	3.633	-332.83%	-330.62%	-273.60%	-246.49%	-226.38%	-168.73%
AODV-3-BeFS-ANS	4.160	-288.41%	-286.51%	-237.28%	-213.76%	-196.39%	-146.63%

Counterexample 2 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Memory Overhead (-) shown as a percentage of Non-inc. SSE Memory					
		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events
BFS-ANS	75.573	-7.08%	-7.03%	-5.91%	-5.34%	-4.92%	-3.79%
DFS-ANS	1.325	-261.93%	-260.38%	-218.77%	-198.45%	-183.14%	-141.20%
AODV-3-BeFS-ANS	0.987	-65.35%	-64.93%	-58.74%	-55.31%	-52.39%	-46.21%

Counterexample 3 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Memory Overhead (-) shown as a percentage of Non-inc. SSE Memory					
		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events
BFS-ANS	74.193	-6.92%	-6.88%	-5.78%	-5.22%	-4.82%	-3.71%
DFS-ANS	1.353	-258.88%	-257.29%	-216.04%	-195.81%	-181.04%	-139.30%
AODV-3-BeFS-ANS	0.970	-66.00%	-65.63%	-59.46%	-56.02%	-53.03%	-46.70%

Counterexample 3 Search Strategy is AODV-3-BeFS-ANS		Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Memory Overhead (-) shown as a percentage of Non-inc. SSE Memory					
N	$MAXDEPTH$		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events
3	15	1.020	-25.71%	-25.48%	-25.46%	-25.49%	-25.34%	-25.34%
4	20	3.577	-9.32%	-8.36%	-8.25%	-8.20%	-8.07%	-7.96%
5	25	35.723	-2.70%	-2.43%	-2.21%	-2.19%	-1.95%	-1.73%

Table 8.3: AODV case study: Average memory consumption (MB) for the non-incremental state space exploration for the experiments shown in Table 8.1. The overhead (-) of the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration memory consumption.

the same, we ensure that the samples are synchronized (i.e., the part of the state space explored at each sample in a non-incremental state space exploration is the same as that at the corresponding sample in the corresponding incremental state space exploration). Finally, since running garbage collection before each sample and measuring the memory take time, we run each replication twice; once to measure time with the memory sampling mechanism disabled and once to measure memory with the memory sampling mechanism enabled.

As shown in Table 8.3, for the largest average memory consumption of the non-incremental state space exploration technique, the largest corresponding memory overhead of the incremental state space exploration technique is only 7.08% of the average non-incremental state space exploration memory consumption. It should also be noted that the extra memory overhead of the incremental state space exploration technique decreases with the increase in q. This is because we do not insert the information of the modified events in $INPUT$; hence, the larger the value of q the smaller the size of $INPUT$.

In Table 8.1, the state space explorer terminates state space exploration if an assertion violation

is detected. Table 8.4 gives the results for Counterexamples 2 and 3 in the case where state space exploration does *not* terminate when an assertion violation is detected. Instead, the state space explorer continues exploring the state space till the maximum specified depth $MAXDEPTH$, but does not explore events from those states that violate the assertion. In general, the savings in Table 8.4 are larger than their counterparts in Table 8.1 because of the reduction in t, which is the proportion of explored events that are tree, as indicated in the last column of Table 8.4. Intuitively, t is higher when the state space explorer terminates state space exploration if an assertion violation is detected (Table 8.1) because the tree events are the ones that "make change" and cause the assertion to be violated. This is especially true in the case of the AODV-3-BeFS-ANS strategy (see Table 8.1) because the goal of the best-first search is to guide the state space explorer towards paths that lead to the assertion violation in less time. As shown in Table 8.4, for the largest non-incremental state space exploration time, the incremental state space exploration technique was able to provide a time saving of 75.25% (or equivalently a 4.04x speedup) with one code change (i.e., one modified event) that does not incur a behavioral change.

In Table 8.5, we measure the memory overhead of each of the incremental state space exploration experiments shown in Table 8.4 and report it as a percentage of the corresponding average non-incremental state space exploration memory consumption. As shown in Table 8.5, for the largest average memory consumption of the non-incremental state space exploration technique, the largest corresponding memory overhead of the incremental state space exploration technique is only 15.10% of the average non-incremental state space exploration memory consumption. Similar to our observations on Table 8.3, the extra memory overhead of the incremental state space exploration technique decreases with the increase in q. This is because we do not insert the information of the modified events in $INPUT$; hence, the larger the value of q the smaller the size of $INPUT$. Note however that the value of the extra memory overhead of the incremental state space exploration technique, not as a percentage but as the product of the percentage given in Table 8.5 and the corresponding average non-incremental state space exploration memory consumption, is almost constant in each column. This is because the size of the state space information stored in $INPUT$ is almost the same in each column because in each experiment the state space explorer does not terminate when an assertion violation is detected but continues exploring the state space till the maximum specified depth $MAXDEPTH$ as we explained above.

Counterexample 2 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time Savings (+) or Overhead (-) shown as a percentage of Non-inc. SSE Time						
		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events	t
BFS-ANS	124.656	74.04%	73.06%	49.62%	39.50%	32.43%	13.23%	0.0698
DFS-ANS	128.636	76.37%	75.47%	51.53%	41.11%	33.99%	14.56%	0.0704
AODV-3-BeFS-ANS	128.365	75.73%	74.86%	50.40%	39.61%	32.44%	11.81%	0.0706
Counterexample 3 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time Savings (+) or Overhead (-) shown as a percentage of Non-inc. SSE Time						
		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events	t
BFS-ANS	121.825	74.00%	73.09%	49.32%	39.57%	32.00%	13.18%	0.0696
DFS-ANS	126.452	76.20%	75.34%	51.29%	40.52%	32.61%	12.24%	0.0703
AODV-3-BeFS-ANS	129.367	76.24%	75.25%	51.09%	40.18%	33.99%	13.34%	0.0705

Table 8.4: AODV case study: Average time (sec.) for non-incremental state space exploration. The savings (+) or overhead (-) of the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. The ad-hoc network consists of N nodes arranged in a chain network topology. The state space explorer does NOT terminate state space exploration if an assertion violation is detected. $p = 0.999$. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

Counterexample 2 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Memory Overhead (-) shown as a percentage of Non-inc. SSE Memory					
		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events
BFS-ANS	75.572	-15.10%	-15.00%	-12.45%	-11.26%	-10.34%	-7.76%
DFS-ANS	3.642	-332.66%	-330.46%	-273.42%	-246.30%	-226.23%	-168.56%
AODV-3-BeFS-ANS	4.185	-286.75%	-284.86%	-235.93%	-212.53%	-195.27%	-145.78%
Counterexample 3 $N = 3$ $MAXDEPTH = 9$ Search Strategy	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Memory Overhead (-) shown as a percentage of Non-inc. SSE Memory					
		No Modified Events	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	Five Modified Events
BFS-ANS	74.192	-15.19%	-15.09%	-12.52%	-11.32%	-10.40%	-7.80%
DFS-ANS	3.534	-338.61%	-336.36%	-278.15%	-250.62%	-230.27%	-171.42%
AODV-3-BeFS-ANS	4.105	-288.73%	-286.83%	-237.44%	-213.94%	-196.62%	-146.67%

Table 8.5: AODV case study: Average memory consumption (MB) for the non-incremental state space exploration for the experiments shown in Table 8.4. The overhead (-) of the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration memory consumption.

Code changes that incur behavioral changes:

To further evaluate the incremental state space exploration technique, we evaluate its performance in another scenario; namely, one in which the implementation of a modified event *does* change. Practically, this corresponds to code changes that do incur behavioral changes. Specifically, we simulate the behavior of a user who tries to implement the route timeout event T_3 correctly. First, the user implements the route timeout event T_3 by deleting an RTE instead of invalidating it causing Counterexample 3 to occur (we call this implementation Version C), then the user figures out that the RTE has to be invalidated instead of being deleted but forgets to increment the destination sequence number causing Counterexample 2 to occur (we call this implementation Version B). Following that, the user figures out the correct implementation, which includes invalidating the RTE *and*

incrementing the destination sequence number (we call this implementation Version A). Table 8.6 shows the state space exploration time under this scenario using both the non-incremental and incremental techniques. We also show the results for the case where the implementation changes from Version B, to C and finally to A. (In fact, we have also considered the case where the implementation changes from Version A, to B and finally to C, and the case where the implementation changes from Version A, to C and finally to B, but we do not report the results for these cases because they are similar to the results for the cases reported here.)

We distinguish between two cases: Case I, which we call WriteEachVersion, and Case II, which we call WriteFirstVersion. In Case I, storing the state space graph in the $OUTPUT$ data structure and writing the $OUTPUT$ data structure to an output file occurs in each version using the incremental state space exploration technique. Specifically, $read = false$ and $write = true$ in the first version while $read = true$ and $write = true$ in the second and third versions. On the other hand, in Case II, storing the state space graph in the $OUTPUT$ data structure and writing the $OUTPUT$ data structure to an output file occurs in only the first version using the incremental state space exploration technique. Specifically, $read = false$ and $write = true$ in the first version while $read = true$ and $write = false$ in the second and third versions.

As shown in Table 8.6, incremental state space exploration is able to provide up to 42.85% overall time savings (or equivalently 1.75x speedup) with one code change (i.e., one modified event) that does incur a behavioral change. The performance results in Case II are better than those in Case I because the second and third versions of Case II avoid the operations associated with $write = true$; namely, inserting event information in $OUTPUT$ and writing the state space graph to the output file. Note that the values of p in Case II are very close to the corresponding ones in Case I; i.e., no significant harm was done by setting $write = false$ in the second and third versions of Case II. In both Cases I and II, the performance results under the scenario B $\rightarrow$ C $\rightarrow$ A are very close to the corresponding ones under the scenario C $\rightarrow$ B $\rightarrow$ A. This is due to the observation that the values of p and q in the former scenario are very close to the corresponding ones in the latter.

It is also interesting to see how the time savings dropped from 73.60% (Table 8.1, first row, case of one modified event) to 68.56% (Table 8.6, case of WriteFirstVersion, second-to-last row) although the value of $q = 0.0108$ is the same in both cases (see Table 8.2) and the average non-incremental state space exploration time is roughly the same in both cases. This reduction in the time savings is due to the reduction in the value of p from $p = 0.999$ (Table 8.1) to $p = 0.9404$ (Table 8.6).

Table 8.7 shows the average memory consumption (MB) for the non-incremental and incremental

Case I (WriteEachVersion): In incremental state space exploration, $read$ and $write$ are set as follows:
$read{=}false$, $write{=}true$ (1^{st} version in each set).
$read{=}true$, $write{=}true$ (2^{nd} and 3^{rd} versions in each set). Same data structure is used for $INPUT$ and $OUTPUT$.

C → B → A	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case I)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version C (Counterexample 3)	121.738	133.209	-9.42%	v = 0.999960 No speedup if $read{=}false$
Version B (Counterexample 2)	123.976	40.375	67.43%	p = 0.9403, q = 0.0108
Version A (No Counterexample)	123.646	40.293	67.41%	p = 0.9404, q = 0.0108
Sum of Versions C, B, and A	369.360	213.877	42.10%	**42.10% overall savings**

B → C → A	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case I)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version B (Counterexample 2)	123.976	136.166	-9.83%	v = 0.999972 No speedup if $read{=}false$
Version C (Counterexample 3)	121.738	38.563	68.32%	p = 0.9521, q = 0.0109
Version A (No Counterexample)	123.646	40.406	67.32%	p = 0.9404, q = 0.0108
Sum of Versions B, C, and A	369.360	215.135	41.75%	**41.75% overall savings**

Case II (WriteFirstVersion): In incremental state space exploration, $read$ and $write$ are set as follows:
$read{=}false$, $write{=}true$ (1^{st} version in each set).
$read{=}true$, $write{=}false$ (2^{nd} and 3^{rd} versions in each set).

C → B → A	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case II)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version C (Counterexample 3)	121.738	133.236	-9.44%	v = 0.999960 No speedup if $read{=}false$
Version B (Counterexample 2)	123.976	38.914	68.61%	p = 0.9403, q = 0.0108
Version A (No Counterexample)	123.646	38.942	68.50%	p = 0.9403, q = 0.0108
Sum of Versions C, B, and A	369.360	211.092	42.85%	**42.85% overall savings**

B → C → A	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case II)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version B (Counterexample 2)	123.976	136.232	-9.89%	v = 0.999972 No speedup if $read{=}false$
Version C (Counterexample 3)	121.738	37.183	69.46%	p = 0.9521, q = 0.0109
Version A (No Counterexample)	123.646	38.875	68.56%	p = 0.9404, q = 0.0108
Sum of Versions B, C, and A	369.360	212.290	42.52%	**42.52% overall savings**

Table 8.6: AODV case study: Time (sec.) for both the non-incremental and the incremental state space exploration techniques. The ad-hoc network consists of 3 nodes arranged in a chain network topology. $MAXDEPTH = 9$. Search strategy is BFS-ANS and the number of replications is 10. The state space explorer does NOT terminate state space exploration if an assertion violation is detected. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

state space explorations for the experiments shown in Table 8.6. As shown in Table 8.7 for both the WriteEachVersion and WriteFirstVersion cases, the average memory consumption of the non-incremental state space exploration is roughly the same in the three versions. This is because the state space explorer does not terminate state space exploration if an assertion violation is detected as explained above. On the other hand, the extra memory overhead incurred by the incremental state space exploration, which is also reported as a percentage (last column) of the corresponding average non-incremental state space exploration memory consumption, increases from one version to another in the WriteEachVersion case more than it does in the WriteFirstVersion case. This is because $write = true$ in the second and third versions in the WriteEachVersion case but $write = false$ in the second and third versions in the WriteFirstVersion case. In summary, WriteFirstVersion provided larger

Case I (WriteEachVersion): In incremental state space exploration, *read* and *write* are set as follows:
read=*false*, *write*=*true* (1^{st} version in each set).
read=*true*, *write*=*true* (2^{nd} and 3^{rd} versions in each set). Same data structure is used for $INPUT$ and $OUTPUT$.

C → B → A	Non-inc. SSE Memory (MB) (*read*=*false*, *write*=*false*)	Incremental SSE Memory (MB) (see caption of Case I)	Incremental SSE Memory Overhead (-)
Version C (Counterexample 3)	74.193	77.657	-4.67%
Version B (Counterexample 2)	75.572	86.873	-14.95%
Version A (No Counterexample)	75.569	87.582	-15.90%
Maximum Memory Overhead of Incremental SSE in Versions C, B, and A			**15.90% maximum overhead**

B → C → A	Non-inc. SSE Memory (MB) (*read*=*false*, *write*=*false*)	Incremental SSE Memory (MB) (see caption of Case I)	Incremental SSE Memory Overhead (-)
Version B (Counterexample 2)	75.572	79.064	-4.62%
Version C (Counterexample 3)	74.193	85.647	-15.44%
Version A (No Counterexample)	75.569	87.582	-15.90%
Maximum Memory Overhead of Incremental SSE in Versions B, C, and A			**15.90% maximum overhead**

Case II (WriteFirstVersion): In incremental state space exploration, *read* and *write* are set as follows:
read=*false*, *write*=*true* (1^{st} version in each set).
read=*true*, *write*=*false* (2^{nd} and 3^{rd} versions in each set).

C → B → A	Non-inc. SSE Memory (MB) (*read*=*false*, *write*=*false*)	Incremental SSE Memory (MB) (see caption of Case II)	Incremental SSE Memory Overhead (-)
Version C (Counterexample 3)	74.193	77.657	-4.67%
Version B (Counterexample 2)	75.572	86.767	-14.81%
Version A (No Counterexample)	75.569	86.764	-14.81%
Maximum Memory Overhead of Incremental SSE in Versions C, B, and A			**14.81% maximum overhead**

B → C → A	Non-inc. SSE Memory (MB) (*read*=*false*, *write*=*false*)	Incremental SSE Memory (MB) (see caption of Case II)	Incremental SSE Memory Overhead (-)
Version B (Counterexample 2)	75.572	79.065	-4.62%
Version C (Counterexample 3)	74.193	85.532	-15.28%
Version A (No Counterexample)	75.569	86.908	-15.00%
Maximum Memory Overhead of Incremental SSE in Versions B, C, and A			**15.28% maximum overhead**

Table 8.7: AODV case study: Average memory consumption (MB) for the non-incremental and incremental state space explorations for the experiments shown in Table 8.6.

savings in time (Table 8.6) and a smaller overhead in memory (Table 8.7) than WriteEachVersion.

8.2 Directed Diffusion Case Study (Revisited)

We apply the non-incremental and the incremental state space exploration procedures to the J-Sim simulation model of the Directed Diffusion [48] data dissemination protocol for wireless sensor networks (Section 5.1).

The integer array representation of a state depends on the protocol-specific information such as the packet headers and payloads, and each node's interest cache entries and data cache entries. The BeFS heuristic that we consider in the experiments is DD-4-BeFS.

We conduct the experiments in this section on a 4 CPU Intel Xeon 3.2 GHz machine running Linux version 2.6.9 with 4 GB memory. We use Sun's 1.5.0 04-b05 Java HotSpot^{TM} Server VM with 0.5 GB initial heap size and 2.5 GB maximum heap size.

8.2.1 Experiments

Code changes that do not incur behavioral changes:

We first study the performance of the incremental state space exploration procedure in three different scenarios: (i) a best-case scenario ($q = 0$): no events are modified, and (ii) two practical-case scenarios ($0 < q < 1$): one event (the packet delivery event T_4) or two events (T_2 and T_4) are modified. We compare between the non-incremental and the incremental state space exploration procedures in each of these three scenarios using the randomized search strategy DD-4-BeFS-ANS. To isolate the savings gained (or overhead incurred) by the incremental state space exploration procedure, the modified events, if any, are just "flagged" as modified but their implementations are not changed. Again, this corresponds to code changes that do *not* incur any behavioral changes.

Table 8.8 gives the time needed to find an assertion violation (Counterexample 1 or 2 as explained in Section 5.3) for different values of N and $MAXDEPTH$, where N is the number of nodes in the wireless sensor network. In Table 8.8, "Non-inc. SSE" refers to Figure 7.3 with $read = false$ and $write = false$, while "Incremental SSE" refers to Figure 7.3 with $read = true$ and $write = false$. Before running the "Incremental SSE", we had to execute Figure 7.3 with $read = false$ and $write = true$ in order to generate the file that contains the state space graph; however, the time needed for this intermediate step is not reported. For each experiment, we ran 10 replications. Each replication has a different seed. The savings gained (or overhead incurred) by the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. As shown in Table 8.8, the incremental state space exploration technique can indeed provide time savings in the best-case scenario ($q = 0$) and the two practical-case scenarios ($0 < q < 1$). For example, for the largest non-incremental state space exploration time (last row), the incremental state space exploration technique was able to provide a time saving of 47.41% (or equivalently a 1.90x speedup) with one code change (i.e., one modified event) that does not incur a behavioral change.

In order to understand why the incremental state space exploration procedure was successful in providing time savings, we show in Table 8.9 a breakdown of the average state space exploration time spent in some selected operations taking the last row in Table 8.8 as an example. As shown in Table 8.9, the costs of executing events (i.e., the sum of copying from the verification model to the simulation model, executing the event handlers in the simulation model, and copying from the simulation model to the verification model), computing hash codes, and checking the assertions are

Search Strategy is DD-4-BeFS-ANS			Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time Savings (+) or Overhead (-) shown as a percentage of Non-inc. SSE Time			
N	$MAXDEPTH$	Counterexample		No Modified Events	One Modified Event	Two Modified Events	t
4	15	1	5.159	52.74%	36.85%	28.14%	0.1888
5	20	1	13.397	64.37%	48.78%	32.44%	0.1927
6	25	1	60.453	75.03%	56.23%	37.98%	0.1693
4	20	2	174.896	72.05%	47.41%	37.08%	0.0972

Table 8.8: Directed diffusion case study: Average time (sec.) for non-incremental state space exploration. The savings (+) or overhead (-) of the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. The wireless sensor network consists of N nodes arranged in a chain network topology. The state space explorer terminates state space exploration if an assertion violation is detected. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

considerably high taking together more than 74% of the total average time in the non-incremental state space exploration. In contrast, the times spent in these three operations in the incremental state space exploration procedure are smaller than their counterparts in the non-incremental exploration. Furthermore, other operations that are only done in the incremental state space exploration procedure (e.g., reading from the input file and searching in $INPUT$) take a comparatively small amount of time (less than 16% of the incremental state space exploration time). Note that in this particular example, $t = 9.72\%$; i.e., the proportion of events that are tree is small. Furthermore, $p = 0.999$; i.e., almost all of the searches in $INPUT$ are successful. All these observations explain why incremental state space exploration procedure provided time savings.

It should also be noticed that the times of the operations that do not appear in the simplified necessary condition (e.g., inserting the hash code of a state in *AlreadyVisited*, inserting a state in *ToBeExplored*, searching in *AlreadyVisited*, computing the BeFS tuple, and generating the enabled events) are almost equal in both the non-incremental and incremental state space exploration techniques. Furthermore, the sum of the times to search in *AlreadyVisited*, compute the BeFS tuple, insert a hash code in *AlreadyVisited*, and insert a state in *ToBeExplored* is of the same order as the sum of the times to search in $INPUT$ and read from the input file in the incremental state space exploration technique. This justifies the assumption that we made in Section 7.3.2.

In Table 8.10, we measure the memory overhead for each of the incremental state space exploration experiments shown in Table 8.8 and report it as a percentage of the corresponding average non-incremental state space exploration memory consumption. As shown in Table 8.10, for the largest average memory consumption of the non-incremental state space exploration technique (last

Counterexample 2 $N = 4$ $MAXDEPTH = 20$ DD-4-BeFS-ANS Search Strategy Operation	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Time (sec.)		
		No Modified Events q=0	One Modified Event q=0.2919	Two Modified Events q=0.4546
From V model to S model	18.787	2.288	7.409	9.936
Execute event handlers	57.945	7.528	26.895	33.053
From S model to V model	16.513	1.844	7.041	9.210
Compute a hash code	15.379	0.019	4.748	7.177
Verify an assertion	22.468	0.000	7.331	10.505
Read from input file	0.000	2.180	1.961	2.169
Search in $INPUT$	0.000	5.527	3.822	3.011
Insert in $AlreadyVisited$	1.196	1.367	1.187	1.172
Search in $AlreadyVisited$	3.130	2.921	2.976	3.012
Generate enabled events	2.852	3.499	3.036	2.858
Insert in $ToBeExplored$	5.161	6.197	5.530	5.464
Compute the BeFS tuple	0.906	0.965	0.872	0.869
Subtotal	144.337	34.335	72.808	88.436
Other operations	30.559	14.547	19.172	21.610
Total time	174.896	48.882	91.980	110.046
Speedup		3.58x	1.90x	1.59x
Number of executed events	1293732	124492	491101	649234

Table 8.9: Directed diffusion case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations. The example shown corresponds to the last row in Table 8.8. $p = 0.9999$

Search Strategy is DD-4-BeFS-ANS			Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE ($read$=$true$, $write$=$false$) Memory Overhead (-) shown as a percentage of Non-inc. SSE Memory		
N	$MAXDEPTH$	Counterexample		No Modified Events	One Modified Event	Two Modified Events
4	15	1	9.392	-11.36%	-10.44%	-9.60%
5	20	1	34.772	-6.10%	-5.38%	-4.57%
6	25	1	158.955	-5.13%	-4.38%	-3.48%
4	20	2	317.640	-9.72%	-7.98%	-6.99%

Table 8.10: Directed diffusion case study: Average memory consumption (MB) for the non-incremental state space exploration for the experiments shown in Table 8.8. The overhead (-) of the incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration memory consumption.

row), the largest corresponding memory overhead of the incremental state space exploration technique is only 9.72% of the average non-incremental state space exploration memory consumption. Similar to our observations on Tables 8.3-8.5, the extra memory overhead of the incremental state space exploration technique decreases with the increase in q. This is because we do not insert the information of the modified events in $INPUT$; hence, the larger the value of q the smaller the size of $INPUT$.

Code changes that incur behavioral changes:

To further evaluate the incremental state space exploration technique, we evaluate its performance in another scenario; namely, one in which the implementation of a new event is added. Similar to Section 8.1.1, this corresponds to code changes that do incur behavioral changes. However, in

	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (Write strategy is WriteFirstVersion)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Example 1: Version B_1 includes the events T_0, T_1, T_3, T_4, and T_5. Version A_1 adds T_2.				
Version B_1	20.373	21.937	-7.68%	v = 1.0 No speedup if $read{=}false$
Version A_1	30.228	16.461	45.54%	p = 0.7505, q = 0.1321
Sum of Versions B_1 and A_1	50.601	38.398	24.12%	**24.12% overall savings**
Example 2: Version B_2 includes the events T_0, T_1, T_4, and T_5. Version A_2 adds T_2.				
Version B_2	18.933	20.532	-8.45%	v = 1.0 No speedup if $read{=}false$
Version A_2	27.732	15.093	45.58%	p = 0.7545, q = 0.1326
Sum of Versions B_2 and A_2	46.665	35.625	23.66%	**23.66% overall savings**
Example 3: Version B_3 includes the events T_0, T_1, T_2, T_3, and T_4. Version A_3 adds T_5.				
Version B_3	18.291	19.337	-5.72%	v = 1.0 No speedup if $read{=}false$
Version A_3	30.392	18.534	39.02%	p = 0.7623, q = 0.2783
Sum of Versions B_3 and A_3	48.683	37.871	22.21%	**22.21% overall savings**
Example 4: Version B_4 includes the events T_0, T_1, T_2, and T_4. Version A_4 adds T_5.				
Version B_4	16.485	17.628	-6.93%	v = 1.0 No speedup if $read{=}false$
Version A_4	27.707	17.558	36.63%	p = 0.7576, q = 0.2866
Sum of Versions B_4 and A_4	44.192	35.186	20.38%	**20.38% overall savings**

Table 8.11: Directed diffusion case study: Time (sec.) for both the non-incremental and the incremental state space exploration techniques. The wireless sensor network consists of 4 nodes arranged in a chain network topology. $MAXDEPTH = 10$. Search strategy is BFS-ANS and the number of replications is 10. No assertion violations. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

Section 8.1.1, the change was a modification of an existing behavior, but in this section, the change is an addition of a new behavior.

We study four examples that are explained in Table 8.11. As indicated in Table 8.11, the incremental state space exploration technique was able to provide up to 24.12% overall time savings (or equivalently 1.32x speedup) with one code change (i.e., one added event) that does incur a behavioral change. The corresponding memory overhead of the incremental state space exploration technique is only 8.10% of the average non-incremental state space exploration memory consumption as shown in the corresponding row in Table 8.12.

8.3 ARQ Case Study (Revisited)

We apply the non-incremental and the incremental state space exploration procedures to the J-Sim simulation model of the Automatic Repeat reQuest (ARQ) protocol (Section 6.1).

The integer array representation of a state depends on the protocol-specific information such as the sequence numbers in the headers of the data and ACK packets, the values of *SeqNoSent*, *SeqNoExpected*, *NumDistinctDataMsgSent*, and *NumDistinctDataMsgReceived*.

We conduct the experiments in this section using the same machine and settings that we used in the directed diffusion case study (Section 8.2).

	Non-inc. SSE Memory (MB) ($read{=}false$, $write{=}false$)	Incremental SSE Memory (MB) (Write strategy is WriteFirstVersion)	Incremental SSE Memory Overhead (-)
Example 1: Version B_1 includes the events T_0, T_1, T_3, T_4, and T_5. Version A_1 adds T_2.			
Version B_1	34.328	35.667	-3.90%
Version A_1	47.666	51.527	-8.10%
Maximum Memory Overhead of Incremental SSE in Versions B_1 and A_1			**8.10% maximum overhead**
Example 2: Version B_2 includes the events T_0, T_1, T_4, and T_5. Version A_2 adds T_2.			
Version B_2	31.676	32.958	-4.05%
Version A_2	44.141	47.710	-8.09%
Maximum Memory Overhead of Incremental SSE in Versions B_2 and A_2			**8.09% maximum overhead**
Example 3: Version B_3 includes the events T_0, T_1, T_2, T_3, and T_4. Version A_3 adds T_5.			
Version B_3	36.507	37.599	-2.99%
Version A_3	47.666	51.085	-7.17%
Maximum Memory Overhead of Incremental SSE in Versions B_3 and A_3			**7.17% maximum overhead**
Example 4: Version B_4 includes the events T_0, T_1, T_2, and T_4. Version A_4 adds T_5.			
Version B_4	33.513	34.555	-3.11%
Version A_4	44.141	47.256	-7.06%
Maximum Memory Overhead of Incremental SSE in Versions B_4 and A_4			**7.06% maximum overhead**

Table 8.12: Directed diffusion case study: Average memory consumption (MB) for the non-incremental and incremental state space explorations for the experiments shown in Table 8.11.

8.3.1 Experiments

To evaluate the incremental state space exploration technique, we evaluate its performance in a scenario in which code changes do incur behavioral changes. Specifically, we simulate the behavior of a user who tries to implement the ACK packet delivery event T_1 correctly. First, the user forgets to make the sender check the sequence number in the ACK before sending a data packet causing Counterexample 1, as explained in Section 6.3, to occur (we call this implementation Version B). Following that, the user figures out the correct implementation, which requires checking the sequence number in the ACK to determine whether a new data packet or a retransmission should be sent (we call this implementation Version A).

Table 8.13 shows the state space exploration time under this scenario using both the non-incremental and incremental techniques. For each experiment, we ran 100 replications. Each replication has a different seed. We distinguish between the two cases: Case I (WriteEachVersion) and Case II (WriteFirstVersion) as explained in Section 8.1.1. As shown in Table 8.13, incremental state space exploration is *not* able to provide any savings in the state space exploration time.

In order to understand why the incremental state space exploration procedure did not provide any time savings, we show in Table 8.14 a breakdown of the average state space exploration time spent in some selected operations taking the second-to-last row in Table 8.13 as an example. As shown in Table 8.14, the costs of executing events (i.e., the sum of copying from the verification model to the simulation model, executing the event handlers in the simulation model, and copying from the simulation model to the verification model), computing hash codes, and checking the assertion are

Case I (WriteEachVersion): In incremental state space exploration, *read* and *write* are set as follows:
$read{=}false$, $write{=}true$ (1^{st} version in each set).
$read{=}true$, $write{=}true$ (2^{nd} version in each set). Same data structure is used for $INPUT$ and $OUTPUT$.

B → A $MAXDEPTH = 30$	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case I)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version B (Counterexample 1)	7.478	8.283	-10.76%	v = 0.923905 No speedup if $read{=}false$
Version A (Correct T_1)	10.111	11.542	-14.16%	p = 0.0500, q = 0.2498
Sum of Versions B and A	17.589	19.825	-12.71%	**12.71% overall overhead**
B → A $MAXDEPTH = 35$	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case I)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version B (Counterexample 1)	22.351	24.304	-8.74%	v = 0.929023 No speedup if $read{=}false$
Version A (Correct T_1)	30.134	34.742	-15.29%	p = 0.0239, q = 0.2499
Sum of Versions B and A	52.485	59.046	-12.50%	**12.50% overall overhead**

Case II (WriteFirstVersion): In incremental state space exploration, *read* and *write* are set as follows:
$read{=}false$, $write{=}true$ (1^{st} version in each set).
$read{=}true$, $write{=}false$ (2^{nd} version in each set).

B → A $MAXDEPTH = 30$	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case II)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version B (Counterexample 1)	7.478	8.283	-10.76%	v = 0.923905 No speedup if $read{=}false$
Version A (Correct T_1)	10.111	10.495	-3.80%	p = 0.0500, q = 0.2498
Sum of Versions B and A	17.589	18.778	-6.76%	**6.76% overall overhead**
B → A $MAXDEPTH = 35$	Non-inc. SSE Time (sec.) ($read{=}false$, $write{=}false$)	Incremental SSE Time (sec.) (see caption of Case II)	Time Savings (+) or Overhead (-)	Observations on Incremental SSE
Version B (Counterexample 1)	22.351	24.304	-8.74%	v = 0.929023 No speedup if $read{=}false$
Version A (Correct T_1)	30.134	31.581	-4.80%	p = 0.0239, q = 0.2499
Sum of Versions B and A	52.485	55.885	-6.48%	**6.48% overall overhead**

Table 8.13: ARQ case study: Time (sec.) for both the non-incremental and the incremental state space exploration techniques. Search strategy is BFS-ANS. The state space explorer does NOT terminate state space exploration if an assertion violation is detected. *AlreadyVisited* stores the hash codes of the states that have already been visited. The stateful search depends on the equality of the hash codes.

not high taking together less than 51% of the total average time in the non-incremental state space exploration. Furthermore, due to a relatively large value of $q = 0.2499$, an extremely small value of $p = 0.0239$, and a relatively large value of $t = 0.2517$, more than 98% of the events are executed in the incremental state space exploration procedure as shown in the last row of Table 8.14. The measured estimates of the other values that appear in the third necessary condition (Section 7.3.2) in a randomly chosen run, which would correspond to a small pilot study performed by the user to provide a preliminary check on the effectiveness of incremental state space exploration, are as follows: $X = 31.08\ \mu s$, $H = 11.88\ \mu s$, $Y = 5.55\ \mu s$, $L = 8.77\ \mu s$, and $F = 2.95\ \mu s$. Clearly, the third necessary condition is violated in this study. The J-Sim state space explorer outputs a message informing the user that the third necessary condition is violated; hence, using the incremental state space exploration procedure in this case study is discouraged.

Version A Correct T_1 $MAXDEPTH = 35$ BFS-ANS Search Strategy Operation	Non-inc. SSE Time (sec.) ($read$=$false$, $write$=$false$)	Incremental SSE Time (sec.) ($read$=$true$, $write$=$false$)
From V model to S model	2.543	2.563
Execute event handlers	4.770	4.850
From S model to V model	2.449	2.482
Compute a hash code	3.611	3.497
Verify an assertion	1.767	1.760
Read from input file	0.000	0.752
Search in $INPUT$	0.000	0.360
Insert in $AlreadyVisited$	0.792	0.793
Search in $AlreadyVisited$	1.748	1.747
Generate enabled events	1.911	1.954
Insert in $ToBeExplored$	1.071	1.097
Compute the BeFS tuple	0.000	0.000
Subtotal	20.662	21.855
Other operations	9.472	9.726
Total time	30.134	31.581
Speedup		none
Number of executed events	318216	314473

Table 8.14: ARQ case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations. The example shown corresponds to the second-to-last row in Table 8.13.

Case I (WriteEachVersion): In incremental state space exploration, $read$ and $write$ are set as follows:
$read$=$false$, $write$=$true$ (1^{st} version in each set).
$read$=$true$, $write$=$true$ (2^{nd} version in each set). Same data structure is used for $INPUT$ and $OUTPUT$.

B → A $MAXDEPTH = 30$	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE Memory (MB) (see caption of Case I)	Incremental SSE Memory Overhead (-)
Version B (Counterexample 1)	7.459	10.375	-39.09%
Version A (Correct T_1)	8.970	15.065	-67.94%
Maximum Memory Overhead of Incremental SSE in Versions B and A			**67.94% maximum overhead**

B → A $MAXDEPTH = 35$	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE Memory (MB) (see caption of Case I)	Incremental SSE Memory Overhead (-)
Version B (Counterexample 1)	22.665	31.310	-38.14%
Version A (Correct T_1)	27.921	47.675	-70.75%
Maximum Memory Overhead of Incremental SSE in Versions B and A			**70.75% maximum overhead**

Case II (WriteFirstVersion): In incremental state space exploration, $read$ and $write$ are set as follows:
$read$=$false$, $write$=$true$ (1^{st} version in each set).
$read$=$true$, $write$=$false$ (2^{nd} version in each set).

B → A $MAXDEPTH = 30$	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE Memory (MB) (see caption of Case II)	Incremental SSE Memory Overhead (-)
Version B (Counterexample 1)	7.459	10.375	-39.09%
Version A (Correct T_1)	8.970	11.856	-32.17%
Maximum Memory Overhead of Incremental SSE in Versions B and A			**39.09% maximum overhead**

B → A $MAXDEPTH = 35$	Non-inc. SSE Memory (MB) ($read$=$false$, $write$=$false$)	Incremental SSE Memory (MB) (see caption of Case II)	Incremental SSE Memory Overhead (-)
Version B (Counterexample 1)	22.665	31.310	-38.14%
Version A (Correct T_1)	27.921	36.383	-30.31%
Maximum Memory Overhead of Incremental SSE in Versions B and A			**38.14% maximum overhead**

Table 8.15: ARQ case study: Average memory consumption (MB) for the non-incremental and incremental state space explorations for the experiments shown in Table 8.14.

Chapter 9

Related Work

The amount of research on verification and validation (V&V) of simulation models is large and considerably focuses on statistical techniques that compare the output of the simulation model with the output of the real system (e.g., the method of simultaneous confidence intervals [8]). In this section, we highlight some previous work that is closely related to ours. For further details on verification, validation, and testing of simulation models, the interested reader is referred to [6].

9.1 Model Checking Network Protocol Implementation Code

Our work on verifying the computerized simulation model of a network protocol using state space exploration and protocol-specific properties is inspired by the previous work on model-checking network protocol implementation code directly for C and C++ (e.g., CMC [67,68] and VeriSoft [38]).

Although CMC has been applied to model-check Linux implementations of networking code (e.g., AODV and TCP), the major distinction between our approach and CMC is that CMC uses *protocol-independent* properties in guiding the best-first search. It does so by attempting to focus on states that are the most different from previously explored states. In contrast, our approach uses *protocol-dependent* properties, inherent to the network protocol and the assertion being checked, to guide a best-first search strategy towards finding an assertion violation faster.

Likewise, VeriSoft uses *protocol-independent* techniques, namely partial-order reduction (POR) using the persistent and sleep sets [38]. POR is an approach towards alleviating the state space explosion problem, and aims to reduce the size of the state space by exploiting the independence relation between events. Independent events can neither disable nor enable each other, and enabled independent events commute; i.e., result in the same state when executed in different orders. POR is a selective search; i.e., at each state s, POR computes a subset τ (called a *persistent set*) of the set of events enabled in s and explores only the events in τ (the other enabled events are not explored).

A subset τ of the set of events enabled in a state s is called persistent in s if all events not in τ that are enabled in s, or in a state reachable from s through events not in τ, are independent with all events in τ. Informally, whatever one does from s, while remaining outside of τ, does not interact with or affect τ.

Traditional algorithms for computing persistent sets exploit information that is typically inferred by a *static* analysis of the code. However, as pointed out in [37], these algorithms suffer from a fundamental limitation, namely determining this information with acceptable precision, in the context of concurrent software systems executing arbitrary code, is often difficult or impossible. In an attempt towards avoiding this inherent imprecision of static analysis, *dynamic* POR [37] was proposed. The algorithm, which is called *dynamic partial-order reduction*, dynamically tracks interactions between processes and then exploits this information to identify backtracking points where alternative paths in the state space need to be explored.

In principle, dynamic POR can be combined with both shuffling and best-first search strategies. Specifically, POR first determines which enabled events to explore, a shuffling procedure then shuffles the sequence in which those events are executed, and a best-first search strategy finally determines which of the successor states being generated can potentially lead to an assertion violation faster.

The idea of using best-first search heuristics to expedite the model checking process has been explored in previous work (e.g., [31, 39, 41, 99, 115]). However, what distinguishes our work from the previous work is that we study the use of protocol-specific heuristics in verifying the simulation model. Moreover, a large part of our work focuses on a specific domain, namely routing and data dissemination protocols for wireless ad-hoc and sensor networks, and attempt to discover effective protocol-specific heuristics that enable a best-first search strategy to find assertion violations in less time and space requirements than classic breadth-first and depth-first search strategies.

9.2 Conventional Explicit-state and Symbolic Model Checking

In contrast to model-checking the implementation code (or the computerized simulation model) of a network protocol directly, conventional explicit-state and symbolic model checkers (e.g., SPIN [47], SMV [64], SAL [106], Murphi [27]) require that the system be first specified using a high-level modeling language. For example, [82] presents the simulation and verification of the priority-ceiling protocol (PCP) using SAL.

In general, the process of describing the system in a high-level modeling language is time-consuming, painstaking, and error-prone [68]. To deal with this problem, there has been some work (e.g., [25, 36, 44, 76]) on translating programming languages (e.g., Java) to the input modeling languages of several conventional model checkers. The idea is to automatically extract an abstract model out of an application written in C or Java and then use conventional model checking to analyze this abstract model. However, this may not be always feasible because some features of C or Java (e.g., memory allocation and bit operations) do not have corresponding ones in the destination modeling language. Therefore, our approach of directly verifying the simulation model, which has to be written by the user anyway for the purpose of performance evaluation, reduces the user's effort and avoids the limitations of the input languages of conventional model checkers.

Java PathFinder (JPF) [52,111] is a state-of-the-art general-purpose explicit-state model checker for Java programs. We gave an overview of JPF in Section 4.4.1 and compared its performance with the J-Sim state space exploration framework in Section 4.4.3. The results of the comparison showed that our state space exploration framework in J-Sim can be significantly faster than that in JPF, with respect to the time needed to find an assertion violation, unless a significant amount of programming effort is done in JPF. As explained in Section 4.5, this result justifies the need for building a framework for the verification of network simulation models in a network simulator instead of using a general-purpose model checking tool since we believe that wireless network protocol designers and simulation modelers will feel more comfortable using a network simulator as a single integrated environment for both building a simulation model and verifying its correctness than using a network simulator for building a simulation model and using another model checking tool (e.g., JPF) for verifying its correctness.

Bhargavan *et al.* provide in [16] a complete automated proof, using the SPIN model checker and the HOL theorem prover, that no routing loops will be formed by AODV if all the nodes in the ad-hoc network (a) always immediately detect when a neighbor restarts its AODV process and the restart is treated as if all links to the neighbor have broken, (b) increment the destination sequence number of a routing table entry to a destination when the route to that destination has expired or broken, and (c) never delete routing table entries. In this thesis, we have shown that our state space exploration framework in J-Sim can discover routing loops caused by violation of each of these three conditions.

9.3 Formal Analysis of Network Simulation

As far as formal analysis of network simulation is concerned, Verisim [14] was developed based on a collection of pre-existing tools, namely ns-2 [103] and the MaC monitoring and checking framework [55]. Verisim replaces the monitor component of MaC by ns-2 and uses the checker component of MaC to verify user-defined properties on *traces* produced by ns-2. It should be noted, however, that not all assertion violations may manifest themselves in a trace because ns-2 does not explore all possible execution paths during a simulation run.

Towards giving formal semantics to simulation models and hence enabling the exploration of several execution paths, a translation from the DEVS (Discrete Event System Specification) [116] modeling paradigm to the Z-DEVS formalism, which combines DEVS, the Z specification language [97] and first order logic, is proposed in [107]. (This is similar to translating programming languages to the input modeling languages of model checkers that was discussed above.) The static properties of the simulation model can then be analyzed with the Z/EVES theorem prover [83] while the temporal properties can be analyzed using a model checker.

9.4 Neural Network/Machine Learning Approaches for Validating Simulation Models

In [61], a neural network approach to the validation of simulation models is presented. Specifically, a number of alternative simulation models train a neural network using multiple statistics (e.g., means, variances, autocovariances, etc.). Hence, the neural network learns to identify key features of these statistics to belong to a specific simulation model. Following that, an experiment with the real system is offered to the neural network. The network then outputs a probability vector, indicating for every simulation model the probability that the data comes from the model. This probability vector can be used to distinguish valid from invalid models.

Another machine learning technique can be found in [62] where Martens et al. make use of concepts from machine learning and fuzzy set theory to define a resemblance relation for measuring the degree of similarity between the input-output transformation of a simulation model and the corresponding input-output transformation of the real system.

9.5 Integrated Environments for Verifying Models

Examples of other software tools that provide an *integrated* environment, which allows the user to both verify a model and use it to evaluate/predict performance, are TwoTowers [12] and Maude [22, 63].

Using TwoTowers, the user models a concurrent system as an algebraic term in $EMPA_r$ [11], which is an extension of $EMPA$ (Extended Markovian Process Algebra) [13] allowing for the specification of performance measures based on rewards. TwoTowers has three kernels: (a) an integrated kernel can simulate an $EMPA_r$ specification and derive performance measures according to the well-known method of independent replications, (b) a functional kernel generates a labeled transition system (LTS) of the $EMPA_r$ specification, and (c) a performance kernel generates a Markov chain of the $EMPA_r$ specification. A version of the Concurrency Workbench of North Carolina (CWB-NC) [23] can then analyze the LTS using different types of formal verification (e.g., model checking to check temporal properties and state space exploration to check assertions). A Markov Chain Analyzer (MarCA) [98] can conduct stationary and transient performance analysis on the Markov chain where the performance measures are derived using the rewards expressed in the $EMPA_r$ specification. TwoTowers has been used for analyzing several distributed algorithms and networking protocols such as the alternating bit protocol, CSMA/CD, ATM switches, and QoS protocols for Internet audio [1].

Maude [63] is a reflective language and system that supports both equational and rewriting logic specification and programming. Maude can be used to create *executable* specifications for a wide range of applications (e.g., other languages, theorem provers, concurrent systems). In fact, Maude can be used to build language extensions for Maude itself. Particularly, Full Maude is implemented in Maude as an extension of Core Maude[1] [22]. Concurrent object-oriented systems can be specified in Full Maude by means of object-oriented modules, which support objects, messages, classes and inheritance. Object-oriented modules can then be executed and also model-checked with an on-the-fly explicit-state Linear Temporal Logic (LTL) model checker [32]. Furthermore, Real-Time Maude [70] is a language and tool supporting the formal specification and analysis of real-time and hybrid systems. Real-Time Maude is implemented in Maude as an extension of Full Maude and offers a wide range of analysis techniques, including timed rewriting for simulation purposes, state space exploration to check assertions, and time-bounded LTL model checking.

The Real-Time Maude LTL model checker has been previously used in [69] for verifying the

[1] Core Maude is the Maude interpreter implemented in C++ and provides all of Maude's basic functionality.

AER/NCA active network protocol suite [53]. In a more recent case study, Ölveczky and Thorvaldsen [71] use Real-Time Maude to formally model the *Optimally Geographical Density Control (OGDC)* algorithm [117] for wireless sensor networks, use the Real-Time Maude specification to perform all the simulation experiments done by the algorithm developers using ns-2 [103], and perform further formal analyses which are beyond the capabilities of (traditional) simulation tools.

The major difference between our work and tools such as TwoTowers and Maude is that we verify a simulation model that is written in an imperative language (namely, Java) rather than a model that is written in a formal specification language.

9.6 Techniques for Analyzing the Correctness of Evolving Software

In this section, we review the large body of work that is closely related to the goals of the incremental state space exploration (ISSE) technique, namely developing techniques for analyzing the correctness of evolving software. We first review the work in the context of model checking and then discuss the work in the context of software testing.

9.6.1 Model Checking

Starting from the seminal work of Sokolsky and Smolka [96], there have been a few papers proposing techniques to model checking incrementally. The techniques address checking properties for non-recursive abstract models and checking safety properties of recursive software [24, 46]. However, all of these techniques focus on control-intensive properties (e.g., method x must never execute until method y has terminated). The ISSE technique presented in this thesis focuses on data-intensive properties (e.g., the absence of routing loops in AODV) where large concrete states need to be maintained to check properties, and changes are typically made to methods and functions that apply in almost every state. Furthermore, the techniques presented in [24,46,60,96] work on abstract models of programs (e.g., a labeled transition system, a control-flow graph) whereas our work focuses on real code written in imperative programming languages that allow the manipulation of dynamically allocated data. In summary, our focus is on checking data-intensive properties, especially for object-oriented programs. In this domain, state space exploration has three characteristics: (1) program states can grow large (and checking data-intensive properties typically requires entire concrete states and not just abstractions); (2) operations, i.e., methods, apply in almost every state; and (3)

execution of operations, computing the hash codes of states and checking the assertions can take a significant portion of the state space exploration time.

Another related project on incremental computation for explicit state model checking is on incremental heap canonicalization [66]. The idea is to have the model checker execute a heap canonicalization algorithm (e.g., [49]) to transform a state s to a canonical representation that is unique for all the states that are behaviorally equivalent to s but differ only in the memory locations of heap objects. This canonical representation is then inserted in the hash table. The incremental heap canonicalization algorithm would then ensure that small changes to the heap only result in relatively small changes in the canonical representation. This allows the model checker to compute the hash code of a state incrementally by only processing those portions of the state that are modified in a transition. Hence, this project considers incremental computation between a pre-state and a post-state of one transition in a state space exploration and not incremental computation between two consecutive state space explorations.

Recently, there has been some work on incremental conformance testing [33]. In conformance testing of communication protocols, the purpose of the tests is to determine whether an implementation of the protocol conforms to (i.e., has the same input/output behavior as defined by) its specification. In incremental conformance testing [33], the goal is to generate tests that would only test the modified parts of the implementation in order to check that they correctly implement the modified parts of the specification. An underlying assumption is that the parts of the system implementation that correspond to the unmodified parts of the specification have not been changed. Another assumption is that before modifying the system specification, its implementation was tested and found to be conforming to the original specification. This assumption is similar to our assumption that the hash codes of the visited states that violate an assertion are not stored in $OUTPUT$. However, the incremental conformance testing technique makes some other very different assumptions on the software evolution than those considered in our work, like the assumption that not only the specification, but also the implementation can be modeled as a finite state machine.

The notion of *incremental model checking* has also been proposed in the context of bounded model checking [17] that performs checking within a user-specified bound (e.g., bound on the length of the execution). However, the focus in such techniques (e.g., [45]) is on incrementally increasing the bound and not incrementally evolving code as in ISSE.

9.6.2 Software Testing

The main approach to address evolving software in the context of testing is regression testing: it checks that a later version of a software still passes the tests that a previous version passed. Researchers have developed numerous methods to improve the basic regression testing.

What distinguishes regression testing from development testing is that in regression testing, an existing suite of tests may be available for reuse. Since re-executing all tests on the modified version of the software may consume prohibitively large amounts of time and resources, regression test selection [40, 42, 43, 80, 118] chooses to run only some of the existing tests on the new version, thus saving resources and speeding up the testing process by not running all tests. A key challenge is to have safe selection; i.e., guarantee that tests that are not selected cannot reveal errors. (Another challenge is to have precise selection; i.e., guarantee that tests that cannot reveal errors are not selected.) The technique presented in [80] constructs control flow graphs for a procedure or program and its modified version and uses these graphs to safely select tests that execute changed code from the original test suite. Several techniques for test selection [73] record some additional information from previous runs (e.g., code coverage and not only the fact that a test passed or failed) to enable safe selection. Our work on ISSE is partly motivated by test selection techniques because we do not post-process the states generated from the non-modified hit events and we do not re-execute the non-modified hit non-tree or deepest events.

Test prioritization [34, 35, 51, 81] reorders the execution of (all or only selected) tests in order to enhance their ability to meet some performance goal (e.g., reveal errors faster, thus reducing the time that a developer has to wait to find failing tests for program changes). In the contexts of software evolution and regression testing, test prioritization techniques can exploit information (e.g., test coverage) gained from the previous execution of test cases to obtain test case orderings. We could also explore prioritization in the context of state space exploration: if a previous exploration leads to an assertion violation for some execution sequence, then we could try that same sequence (and its neighborhood) in the subsequent exploration. Another possible idea is to use insights gained from the counterexamples discovered while exploring the state space of one version to devise a better state ranking mechanism for exploring the state space of a subsequent version.

Impact analysis [2, 59, 72, 79] finds (statically or dynamically) which code changes could affect which tests, thus aiding test selection. We could leverage impact analysis to improve how ISSE determines what events are modified and need to be executed: ISSE currently uses event-level granularity, but even when the code of some event changes, it does not necessarily mean that the

execution would change for all possible input states.

Automatically decomposing system tests (often used in regression testing) into subsystem or unit tests [74,84] also helps in speeding up regression testing: when a programmer changes some program subsystem/unit, it is not necessary to rerun an entire, potentially time-consuming system test, but it suffices to rerun a focused, rapid subsystem/unit test. For example, the technique presented in [74] allows for (1) selecting a subsystem of interest in a given application, (2) capturing at runtime all the interactions between such subsystem and the rest of the application, and (3) replaying the recorded interactions on the subsystem in isolation. While these approaches are useful in regression testing, it does not seem that they have a direct application in ISSE since executions in ISSE are already fairly short and mostly exercise one program unit.

Chapter 10

Conclusions and Future Work

In this chapter, we conclude the thesis and provide suggestions for potential future research work.

10.1 Conclusions

In this thesis, we present the design, implementation and performance evaluation of a state space exploration (SSE) framework that enriches the set of verification and validation (V&V) software tools available to network simulation modelers and protocol designers. Our framework uses state space exploration to explore the (entire) state space created by a network simulation model and check whether the model satisfies certain assertions that the real network protocol satisfies. We make use of structural properties in the underlying state space of the protocol along two orthogonal dimensions; the first uses a non-trivial *simulation relation* to prune the paths to be searched, and the second is *state ranking* that determines whether a state is "better than" another in order to enable the implementation of a best-first search (BeFS).

We demonstrate the effectiveness of our framework by verifying the simulation models of two widely used and fairly complex network protocols: the Ad-hoc On-Demand Distance Vector (AODV) routing protocol for wireless ad-hoc networks and the directed diffusion data dissemination protocol for wireless sensor networks. Moreover, we verify the simulation model of a reliable unicast protocol: the *Automatic Repeat reQuest (ARQ)* protocol. Experimental results show that the state space explorer is able to find violations of an assertion within acceptable time and space requirements. Furthermore, BeFS heuristics that exploit *properties inherent to the network protocol and the assertion being checked* can expedite the process of finding those violations by several orders of magnitude. We study several BeFS heuristics for both AODV and directed diffusion, and provide guidelines for good heuristics based on our results.

We compare the performance of our SSE framework to that of a state-of-the-art model checker for Java programs, namely Java PathFinder (JPF). The results of the comparison show that the

time needed to find an assertion violation by our state space exploration framework in J-Sim can be significantly less than that in JPF unless a significant amount of programming effort is done in JPF to make its performance close to that of our SSE framework.

We also present *incremental* state space exploration (ISSE), a technique that aims to provide a speedup in the state space exploration time of evolving simulation models; i.e., simulation models whose code changes from one version to another. A code change may or may not lead to a behavioral change. We analytically obtain necessary conditions for the ISSE technique to provide a speedup in the state space exploration time when compared to a traditional (non-incremental) SSE technique. We applied the ISSE technique to our case studies. In two case studies (namely AODV and directed diffusion), ISSE provided a speedup whereas in one case study (namely ARQ), it did not provide a speedup because the necessary conditions were violated. The speedups in state space exploration time come at a reasonable cost of memory overhead.

10.2 Future Work

A potential avenue for future work that extends the work in this thesis is to devise efficient heuristics for each class of network protocols (e.g., routing protocols, MAC protocols, and transport protocols, etc.) so that if the simulation model of a protocol belonging to a certain class is to be verified, the user can use the appropriate heuristic for that class without having to start from scratch. In other words, the heuristics will be class-specific instead of protocol-specific. Another avenue of future research is state similarity. In this thesis, we have used only one way of defining similar states. Future work can study the performance of different granularities of state similarity.

Our state space exploration framework in J-Sim can only find violations of assertions. One possible extension to our work is to find violations of other temporal properties (e.g., liveness) or prove the correctness of the simulation model of a network protocol.

Our work on ISSE can also be extended along several directions. One possible idea is to use insights gained from the counterexamples discovered while exploring the state space of one version of the simulation model to devise a better state ranking mechanism for exploring the state space of a subsequent version. Another possible avenue for future work is to study whether or not, and how, ISSE can help improve the scalability of the framework to explore the state spaces created by larger and/or different network topologies. Finally, ISSE currently uses event-level granularity as the unit(s) of evolution. Future work can study the performance of ISSE under different granularities.

Appendix A

Simulation Relations

In this appendix, we argue that each of the simulation relations that we used in our experiments in Chapters 4-5 is indeed a simulation relation.

We first define a *multiset automaton* as a 4-tuple (Q, M, q_0, δ) where:

- Q is a set of states.
- M is a set of messages.
- q_0 is the initial state.
- δ is the transition relation, that is $\delta \subseteq Q \times (\{\tau\} \cup (M \times \{!, ?\})) \times Q$

We define B as a multiset of messages and define $S = Q \times B$. We use the notation in Section 3.2.3.

Based on these definitions, we define the following transitions for $q_1, q_1' \in Q$; $m \in M$; and $B_1, B_1' \in B$:

- *internal transition*: $(q_1, B_1) \xrightarrow{\tau} (q_1', B_1') \iff B_1' = B_1$ and $(q_1, \tau, q_1') \in \delta$
- *receive transition*: $(q_1, B_1) \xrightarrow{m?} (q_1', B_1') \iff B_1' = B_1 \backslash \{m\}$ and $(q_1, m?, q_1') \in \delta$
- *send transition*: $(q_1, B_1) \xrightarrow{m!} (q_1', B_1') \iff B_1' = B_1 \cup \{m\}$ and $(q_1, m!, q_1') \in \delta$

Furthermore, we can define the following composite transitions for $q_1, q_1' \in Q$; $B_1, B_1' \in B$; and $m_0, m_1, m_2, \ldots, m_{n-1}, m_n \in M$ where $n \geq 1$:

- *receive/send transition*: $(q_1, B_1) \xrightarrow{m_0?/m_1!} (q_1', B_1') \iff \exists B_0 \in B$ and $\exists q_0 \in Q$ such that $(q_1, B_1) \xrightarrow{m_0?} (q_0, B_0)$, and $(q_0, B_0) \xrightarrow{m_1!} (q_1', B_1')$. Note that a receive/send transition is defined as a sequence of one receive transition and one send transition.
- *broadcast transition*: $(q_1, B_1) \xrightarrow{\{m_1, m_2, \ldots, m_{n-1}, m_n\}!} (q_1', B_1') \iff \exists B_{0,1}, B_{0,2}, \ldots, B_{0,n-1} \in B$ and $\exists q_{0,1}, q_{0,2}, \ldots, q_{0,n-1} \in Q$ such that $(q_1, B_1) \xrightarrow{m_1!} (q_{0,1}, B_{0,1})$, $(q_{0,1}, B_{0,1}) \xrightarrow{m_2!} (q_{0,2}, B_{0,2})$, $\ldots$, $(q_{0,n-2}, B_{0,n-2}) \xrightarrow{m_{n-1}!} (q_{0,n-1}, B_{0,n-1})$, and $(q_{0,n-1}, B_{0,n-1}) \xrightarrow{m_n!} (q_1', B_1')$. Note that a broadcast transition is defined as a sequence of n send transitions.

- *receive/broadcast transition*: $(q_1, B_1) \xrightarrow{m_0?/\{m_1,m_2,\ldots,m_{n-1},m_n\}!} (q_1', B_1') \iff \exists B_0 \in B$ and $\exists q_0 \in Q$ such that $(q_1, B_1) \xrightarrow{m_0?} (q_0, B_0)$, and $(q_0, B_0) \xrightarrow{\{m_1,m_2,\ldots,m_{n-1},m_n\}!} (q_1', B_1')$. Note that a receive/broadcast transition is defined as a sequence of one receive transition and n send transitions.

Consider the binary relation R over S (i.e., $R \subseteq S \times S$). For $s_1, s_2 \in S$; $q_1, q_2 \in Q$; and $B_1, B_2 \in B$ such that $s_1 = (q_1, B_1)$ and $s_2 = (q_2, B_2)$. We define R as $(s_1, s_2) \in R \iff q_1 = q_2$ and $B_1 \subseteq B_2$.

We consider an assertion ϕ that depends on Q only. We claim that R is a simulation relation; i.e., if $(s_1, s_2) \in R$ then (1) $s_1 \vDash \phi \iff s_2 \vDash \phi$, and (2) for each $e \in (\{\tau\} \cup (M \times \{!, ?\}))$, $s_1 \xrightarrow{e} s_1'$ where $s_1' = (q_1', B_1')$, $q_1' \in Q$, $B_1' \in B$, and $s_1' \in S$ implies that $\exists s_2' = (q_2', B_2')$ such that $q_2' \in Q$, $B_2' \in B$, $s_2' \in S$, $s_2 \xrightarrow{e} s_2'$ and $(s_1', s_2') \in R$. In order to argue that R is a simulation relation, we consider the following three cases:

- Case of an internal transition $e = \tau$: Since $q_1 = q_2$, then $q_1' = q_2'$. Hence, $s_1' \vDash \phi \iff s_2' \vDash \phi$. Since $B_1 \subseteq B_2$, $B_1' = B_1$ and $B_2' = B_2$, then $B_1' \subseteq B_2'$. Therefore, $(s_1', s_2') \in R$.

- Case of a receive transition $e = m?$: Since $q_1 = q_2$, then $q_1' = q_2'$. Hence, $s_1' \vDash \phi \iff s_2' \vDash \phi$. Since $B_1 \subseteq B_2$, $B_1' = B_1 \backslash \{m\}$ and $B_2' = B_2 \backslash \{m\}$, then $B_1' \subseteq B_2'$. Therefore, $(s_1', s_2') \in R$.

- Case of a send transition $e = m!$: Since $q_1 = q_2$, then $q_1' = q_2'$. Hence, $s_1' \vDash \phi \iff s_2' \vDash \phi$. Since $B_1 \subseteq B_2$, $B_1' = B_1 \cup \{m\}$ and $B_2' = B_2 \cup \{m\}$, then $B_1' \subseteq B_2'$. Therefore, $(s_1', s_2') \in R$.

Similar arguments can be made about broadcast and receive/broadcast transitions because they are defined in terms of receive and send transitions.

The simulation models of AODV (Section 4.2) and directed diffusion (Section 5.2) can be easily seen to be instances of the multiset automaton defined above where Q represents the protocol state and the neighborhood information, and M represents the multiset of packets. Furthermore, in both case studies, the event T_0 is a broadcast transition, the events T_1, T_2 and T_3 are internal transitions, the event T_4 is a receive/broadcast transition, and the event T_5 is a receive transition. Moreover, the simulation relations defined in Section 4.2 and Section 5.2 are instances of the simulation relation R defined above. Note that the assertions used in both case studies do not depend on the multiset of packets; i.e., depend on Q only. Hence, each of the simulation relations that we used in our experiments in Chapters 4-5 is indeed a simulation relation.

Appendix B

More Details of the AODV Case Study

In this appendix, we provide the traces and the explanations of the three counterexamples of the AODV case study (Section 4.3).

B.1 Trace and Explanation of Counterexample 1

Using a breadth-first search strategy (BFS-AC), the state space explorer found an assertion violation in the J-Sim simulation model of AODV, whose trace is shown in Figure B.1.

The counterexample, shown in Figure B.1, can be explained as follows. State 1 is the initial state. In state 2, $n0$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. Similarly, in state 3, $n1$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 4, $n1$ receives the RREQ packet sent by $n0$ and since neither does it have a route to the destination nor is it the destination itself, it rebroadcasts the RREQ packet. In state 5, $n2$ receives the RREQ packet sent by $n1$ in state 3 and since it is the destination itself, it responds by unicasting a RREP packet after incrementing $seqno_2$. In state 6, $n2$ receives the RREQ packet sent by $n0$ in state 2 and since it is the destination itself, it responds by unicasting a RREP packet after incrementing $seqno_2$.

In state 7, $n1$ receives the RREP that is destined to $n0$ and forwards it to $n0$. In addition, $n1$ sets up a forward pointer to the node from which the RREP came (i.e., $n2$), thus establishing a valid routing table entry to the destination $n2$ (note the change of the hop count field from ∞ in state 6 to 1 in state 7 and the change of the sequence number field from 0 in state 6 to 6 in state 7). In state 8, $n0$ receives the RREP packet and establishes a valid routing table entry to the destination $n2$. In state 9, the AODV process in $n1$ restarts and in state 10, $n1$ receives the RREP packet that was sent by $n2$ in state 5 in which $seqno_2$ was set to 4. $n1$ establishes a valid routing table entry to the destination $n2$. Nevertheless, as shown in Figure B.1, $nexthop_{0,2} = 1$ but $seqno_{0,2} > seqno_{1,2}$; i.e., a routing loop is created.

It should be noted, however, that if the restart of the AODV process at $n1$ in state 9 was because of a node reboot, the link layer of $n0$ may be able to detect that the link between $n0$ and $n1$ was broken causing the invocation of a *broken link event handler*, and preventing the routing loop from taking place. Specifically, the simulation model of AODV in J-Sim supports a link layer mechanism to detect broken links. The link layer (e.g., IEEE 802.11) detects a broken link (e.g., by the absence of a link layer ACK each time a packet is transmitted to the next hop; or the failure to get a CTS after sending an RTS each time a packet needs to be transmitted to the next hop and the retry count exceeds the maximum retry limit). Upon detecting a broken link, the link layer notifies the AODV process. In turn, the AODV process executes a broken link event handler. If the broken link is closer to the destination than the source, the node attempts a local repair by sending a RREQ to discover a route to the destination; otherwise, the node invalidates the routing table entries to all the destinations that have become unreachable due to the broken link, and broadcasts a route error (RERR) packet announcing the node IDs of all these unreachable destinations.

The simulation model of AODV in J-Sim also supports a network layer mechanism to detect broken links. In the network layer mechanism (which is optional in both the AODV Draft (version 11) [77] and the simulation model of AODV in J-Sim), each node has a *neighbors list* that contains the node IDs of its neighbors. Neighboring nodes exchange HELLO packets to establish and maintain the neighborhood information. Each entry in the neighbors list is associated with a lifetime. On receiving a HELLO packet, a node creates an entry (or refreshes the lifetime of an existing entry if one already exists) for the source node in the neighbors list. Periodically, every $HELLO\_INTERVAL$ seconds, a neighbor timeout event is triggered, causing the deletion of all the entries in the neighbors list that have expired. According to the AODV Draft (version 11) [77], if a node does not receive any packets (HELLO or otherwise) from one of its neighbors for more than $ALLOWED\_HELLO\_LOSS \times HELLO\_INTERVAL$, it should assume that the link to this neighbor is currently lost and invalidate the routing table entries to the destinations that became unreachable because of this broken link and broadcast a RERR packet. However, we discover that the simulation model of AODV in J-Sim does not follow this part of specification. Specifically, in the simulation model of AODV in J-Sim, if a node does not receive any packets (HELLO or otherwise) from one of its neighbors for more than $1.5 \times ALLOWED\_HELLO\_LOSS \times HELLO\_INTERVAL$, it deletes the neighbor's information from the neighbors list *without* invalidating the routing table entries to the destinations that became unreachable because of the broken link or broadcasting a RERR packet. Furthermore, as stated in AODV Draft (version 11) [77], after a node reboot, a node

waits for $DELETE\_PERIOD$. During this time, the node does not transmit any RREP packets, and if it receives a data packet for some other destination, it should broadcast a RERR packet and must reset the waiting timer to expire after the current time plus $DELETE\_PERIOD$. We found that the simulation model of AODV in J-Sim does not follow this part of specification either. It was shown in [15] that by the time the rebooted node comes out of the waiting phase and becomes an active router again, none of its neighbors will be using it as an active next hop any more.

B.2 Trace and Explanation of Counterexample 2

The first error that we manually injected is not to increment the destination sequence number when invalidating a routing table entry. In order not to get the same counterexample shown in Figure B.1, we required that the counterexample should contain at least one state that is generated due to a route timeout event because the route timeout event triggers the invalidation of a routing table entry. Using a depth-first search strategy (DFS-AC), the state space explorer found the assertion violation, whose trace is shown in Figure B.2.

The counterexample, shown in Figure B.2, can be explained as follows. State 1 is the initial state. In state 2, $n1$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 3, $n2$ receives the RREQ packet sent by $n1$ and since it is the destination itself, it responds by unicasting a RREP packet after incrementing $seqno_2$. Furthermore, $n2$ inserts the pair $< n = n1, bid_n = 1 >$ in its broadcast ID cache and establishes a valid reverse route to the requesting node $n1$. In state 4, the single entry in the broadcast ID cache of $n2$ is deleted due to a broadcast ID timeout event. In state 5, $n1$ receives the RREP that was sent by $n2$ in state 3. Consequently, $n1$ sets up a forward pointer to the node from which the RREP came (i.e., $n2$), thus establishing a valid routing table entry to the destination $n2$.

In state 6, $n0$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 7, $n1$ receives the RREQ packet sent by $n0$ and since it has a fresh enough route to the destination $n2$, $n1$ responds by unicasting a RREP packet back to $n0$. In state 8, the route timeout event takes place at $n1$ causing the invalidation of the routing table entry to $n2$ (note the change of the hop count field from 1 in state 7 to ∞ in state 8) without incrementing the destination sequence number. In state 9, $n0$ receives the RREP packet and establishes a valid routing table entry to the destination $n2$. In state 10, $n0$ receives a RREQ packet sent by $n1$ and since it has a fresh enough route to the destination $n2$, it responds by unicasting a RREP packet back to $n1$. In state 11, $n1$

receives the RREP packet and since the destination sequence number field in the RREP packet is equal to the destination sequence number field in the routing table entry to $n2$ and the hop count field in the RREP packet is less than the hop count field in the routing table entry to $n2$, $n1$ accepts the new offered route and establishes a valid routing table entry to the destination $n2$. Nevertheless, as shown in Figure B.2, $nexthop_{0,2} = 1$ and $nexthop_{1,2} = 0$; i.e., a routing loop is created.

B.3 Trace and Explanation of Counterexample 3

The second error that we manually injected is to delete (instead of invalidating) a routing table entry when its lifetime expires. In order not to get the same counterexample shown in Figure B.1, we required that the counterexample should contain at least one state that is generated due to a route timeout event. Using AODV-1-BeFS-AC, the state space explorer found the assertion violation, whose trace is shown in Figure B.3.

States 1-8 of the counterexample can be explained as explained in the previous counterexamples. In state 9, $n0$ receives a RREQ packet sent by $n1$ and since it has a fresh enough route to the destination $n2$, it responds by unicasting a RREP packet. In state 10, the route timeout event takes place at $n1$ causing the deletion of the routing table entry to $n2$. In state 11, $n1$ receives the RREP packet, accepts the offered route and establishes a valid routing table entry to the destination $n2$. Nevertheless, as shown in Figure B.3, $nexthop_{0,2} = 1$ and $nexthop_{1,2} = 0$; i.e., a routing loop is created.

```
Counterexample
State 1 Depth = 0
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network Empty
State 2 Depth = 1
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network RREQ(src:0--dest:1)
State 3 Depth = 2
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:0--dest:1), RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 4 Depth = 3
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=∞ seqno=0 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:2), RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 5 Depth = 4
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=∞ seqno=0 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:2), RREP(src:2--dest:1--seqno:4)
State 6 Depth = 5
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=∞ seqno=0 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1;
               RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4), RREP(src:2--dest:0--seqno:6)
State 7 Depth = 6
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=1 seqno=6 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1;
               RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4), RREP(src:1--dest:0--seqno:6)
State 8 Depth = 7
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=6 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=1 seqno=6 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1;
               RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 9 Depth = 8
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=6 next=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1;
               RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 10 Depth = 9
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=6 next=1
        Node 1 seqno=2; bid=1; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1;
               RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0)
```

Figure B.1: AODV case study: Trace of counterexample 1 obtained using BFS-AC.

```
Counterexample
State 1 Depth = 0
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network Empty
State 2 Depth = 1
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 3 Depth = 2
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 4 Depth = 3
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 5 Depth = 4
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0)
State 6 Depth = 5
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:0--dest:1)
State 7 Depth = 6
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=1 seqno=4 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREP(src:1--dest:0--seqno:4)
State 8 Depth = 7
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=∞ seqno=4 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREP(src:1--dest:0--seqno:4)
State 9 Depth = 8
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=∞ seqno=4 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0)
State 10 Depth = 9
        Node 0 seqno=4; bid=2; BcastID: src=1, id=1;
               RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=∞ seqno=4 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREP(src:0--dest:1--seqno:4)
State 11 Depth = 10
        Node 0 seqno=4; bid=2; BcastID: src=1, id=1;
               RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=3 seqno=4 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network Empty
```

Figure B.2: AODV case study: Trace of counterexample 2 obtained using DFS-AC.

```
Counterexample
State 1 Depth = 0
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network Empty
State 2 Depth = 1
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 3 Depth = 2
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 4 Depth = 3
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0)
State 5 Depth = 4
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0)
State 6 Depth = 5
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:0--dest:1)
State 7 Depth = 6
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=1 seqno=4 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREP(src:1--dest:0--seqno:4)
State 8 Depth = 7
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=1 seqno=4 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0)
State 9 Depth = 8
        Node 0 seqno=4; bid=2; BcastID: src=1, id=1;
               RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=2 hops=1 seqno=4 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREP(src:0--dest:1--seqno:4)
State 10 Depth = 9
        Node 0 seqno=4; bid=2; BcastID: src=1, id=1;
               RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREP(src:0--dest:1--seqno:4)
State 11 Depth = 10
        Node 0 seqno=4; bid=2; BcastID: src=1, id=1;
               RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1;
               RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=3 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network Empty
```

Figure B.3: AODV case study: Trace of counterexample 3 obtained using AODV-1-BeFS-AC.

Appendix C

More Details of the Directed Diffusion Case Study

In this appendix, we provide the traces and the explanations of the two counterexamples of the directed diffusion case study (Section 5.3).

C.1 Trace and Explanation of Counterexample 1

Figure C.1 shows the trace of Counterexample 1, which can be explained as follows. State 1 is the initial state. In state 2, $n0$ initiates a sensing task by broadcasting an INTEREST packet. In state 3, $n1$ receives the INTEREST packet sent by $n0$, sets up an exploratory gradient whose direction is set toward $n0$ (indicated by $prevHop = 0$ $rate = 60.0$ where 60.0 seconds is the data rate of the exploratory gradient) and rebroadcasts the INTEREST packet. In state 4, $n2$ receives the INTEREST packet sent by $n1$, sets up an exploratory gradient whose direction is set toward $n1$ (because this is the node from which the INTEREST message is received) and rebroadcasts the INTEREST packet. In state 5, $n1$ receives the INTEREST packet sent by $n2$ in state 4 and sets up an exploratory gradient whose direction is set toward $n2$. However, $n1$ did not rebroadcast the INTEREST packet because it recently resent a matching interest (in state 3).

In state 6, $n3$ receives the INTEREST packet sent by $n2$, sets up an exploratory gradient whose direction is set toward $n2$ and rebroadcasts the INTEREST packet. In addition, since $n3$ is located in the area specified by the INTEREST packet, it tasks its local sensors to begin collecting samples and sends a DATA packet to each neighbor for whom it has a gradient (in this case, $n3$ has a gradient to $n2$ only). In state 7, $n2$ receives the DATA packet sent by $n3$ in state 6 and, since the received DATA packet does not have a matching data cache entry, $n2$ adds the received DATA packet to the data cache (indicated by $source = 3$ in state 7) and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n2$ has a gradient to $n1$ only). In state 8, $n1$ receives the DATA packet sent by $n2$ in state 7 and, since the received DATA packet does not have a matching data cache entry, $n1$ adds the received DATA packet to the data cache (indicated by $source = 2$ in

state 8) and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n1$ has a gradient to both $n0$ and $n2$). In state 9, $n0$ receives the DATA packet from $n1$. Since $n0$ has received this previously unseen event from $n1$, it positively reinforces $n1$ by sending a positive reinforcement packet (denoted as $POSREINFORCE$) to $n1$. In state 10, $n1$ receives the positive reinforcement packet sent by $n0$. Hence, $n1$ establishes a reinforced gradient whose direction is set toward $n0$ (indicated by $prevHop = 0$ $rate = 5.0$ where 5.0 seconds is the data rate of the reinforced gradient). In turn, $n1$ should positively reinforce its preferred neighbor. Assuming that all nodes use the same data-driven local rule for determining the preferred neighbor, $n1$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n1$'s preferred neighbor is $n2$; hence, $n1$ sends a positive reinforcement packet to $n2$.

In state 11, the data cache timeout event takes place at $n2$ causing the deletion of the data packet received in state 7. In state 12, $n2$ receives the DATA packet sent by $n1$ in state 8 and, since the received DATA packet does not have a matching data cache entry, $n2$ adds the received DATA packet to the data cache (indicated by $source = 1$ in state 12) and resends the DATA packet to each neighbor for whom it has a gradient. In state 13, $n2$ receives the positive reinforcement packet sent by $n1$ in state 10. Hence, $n2$ establishes a reinforced gradient whose direction is set toward $n1$ (indicated by $prevHop = 1$ $rate = 5.0$). In turn, $n2$ should positively reinforce its preferred neighbor. Using the same data-driven local rule for determining the preferred neighbor, $n2$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n2$'s preferred neighbor is $n1$; hence, $n2$ sends a positive reinforcement packet to $n1$. In state 14, $n1$ receives the positive reinforcement packet sent by $n2$ and establishes a reinforced gradient whose direction is set toward $n2$ (indicated by $prevHop = 2$ $rate = 5.0$). However, a loop is created in the reinforced path between $n1$ and $n2$; i.e., ($RPath(1,2) \wedge RPath(2,1)$) violating the loop-free assertion. Furthermore, the positive reinforcement packet did not reach the data source causing a disruption in the reinforced path (i.e., the reinforced path does not include the data source).

C.2 Trace and Explanation of Counterexample 2

Using the same initial state used in Counterexample 1, we show that a node reboot may also create a loop in the reinforced path. In order not to get the same counterexample shown in Figure C.1, we required that the counterexample should contain at least one state that is generated due to a node

reboot event. Furthermore, in order to show that the assertion violation may still take place even if the data cache timeout event does not happen (i.e., the data cache size is infinite), we disable the data cache timeout event.

The counterexample, shown in Figure C.2, can be explained as follows. State 1 is the initial state. In state 2, $n0$ initiates a sensing task by broadcasting an INTEREST packet. In state 3, $n1$ receives the INTEREST packet sent by $n0$, sets up an exploratory gradient whose direction is set toward $n0$ and rebroadcasts the INTEREST packet. In state 4, $n2$ receives the INTEREST packet sent by $n1$, sets up an exploratory gradient whose direction is set toward $n1$ and rebroadcasts the INTEREST packet. In state 5, $n3$ receives the INTEREST packet sent by $n2$, sets up an exploratory gradient whose direction is set toward $n2$ and rebroadcasts the INTEREST packet. In addition, since $n3$ is located in the area specified by the INTEREST packet, it tasks its local sensors to begin collecting samples and sends a DATA packet to each neighbor for whom it has a gradient (in this case, $n3$ has a gradient to $n2$ only).

In state 6, $n2$ receives the INTEREST packet sent by $n3$ in state 5 and sets up an exploratory gradient whose direction is set toward $n3$. However, $n2$ did not rebroadcast the INTEREST packet because it recently resent a matching interest (in state 4). In state 7, $n0$ rebroadcasts the INTEREST packet. In state 8, $n1$ reboots causing the deletion of all the entries in the interest cache. In state 9, $n1$ receives the INTEREST packet sent by $n0$, sets up an exploratory gradient whose direction is set toward $n0$ and rebroadcasts the INTEREST packet. In state 10, $n1$ receives the INTEREST packet sent by $n2$ in state 4 and sets up an exploratory gradient whose direction is set toward $n2$. In state 11, $n2$ receives the DATA packet sent by $n3$ in state 5 and, since the received DATA packet does not have a matching data cache entry, $n2$ adds the received DATA packet to the data cache and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n2$ has a gradient to both $n1$ and $n3$). In state 12, the gradient to $n1$ stored in the interest cache of $n2$ is removed due to a gradient timeout event.

In state 13, $n2$ reboots causing the deletion of all the entries in the data and interest caches. In state 14, $n2$ receives the INTEREST packet sent by $n1$ in state 9, sets up an exploratory gradient whose direction is set toward $n1$ and rebroadcasts the INTEREST packet. In state 15, $n1$ receives the DATA packet sent by $n2$ in state 11 and, since the received DATA packet does not have a matching data cache entry, $n1$ adds the received DATA packet to the data cache and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n1$ has a gradient to both $n0$ and $n2$). In state 16, $n2$ receives the DATA packet sent by $n1$ and, since the received DATA packet

does not have a matching data cache entry, $n2$ adds the received DATA packet to the data cache and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n2$ has a gradient to $n1$ only). In state 17, $n0$ receives the DATA packet from $n1$. Since $n0$ has received this previously unseen event from $n1$, it positively reinforces $n1$ by sending a positive reinforcement packet to $n1$. In state 18, $n1$ receives the positive reinforcement packet sent by $n0$. Hence, $n1$ establishes a reinforced gradient whose direction is set toward $n0$. In turn, $n1$ should positively reinforce its preferred neighbor. Assuming that all nodes use the same data-driven local rule for determining the preferred neighbor, $n1$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n1$'s preferred neighbor is $n2$; hence, $n1$ sends a positive reinforcement packet to $n2$. In state 19, $n2$ receives the positive reinforcement packet sent by $n1$. Hence, $n2$ establishes a reinforced gradient whose direction is set toward $n1$. In turn, $n2$ should positively reinforce its preferred neighbor. Using the same data-driven local rule for determining the preferred neighbor, $n2$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n2$'s preferred neighbor is $n1$; hence, $n2$ sends a positive reinforcement packet to $n1$. In state 20, $n1$ receives the positive reinforcement packet sent by $n2$ and establishes a reinforced gradient whose direction is set toward $n2$. However, similar to counterexample 1, a loop is created in the reinforced path between $n1$ and $n2$; i.e., ($RPath(1,2) \wedge RPath(2,1)$). In addition, the positive reinforcement packet did not reach the data source causing a disruption in the reinforced path (i.e., the reinforced path does not include the data source).

```
Counterexample
State 1 Depth = 0
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network Empty
State 2 Depth = 1
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:0--dest:1)
State 3 Depth = 2
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop = 0 rate = 60.0); Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 4 Depth = 3
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop = 0 rate = 60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) INTEREST(src:2--dest:3)
State 5 Depth = 4
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop = 0 rate = 60.0) (prevHop = 2 rate = 60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:3)
State 6 Depth = 5
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop = 0 rate = 60.0) (prevHop = 2 rate = 60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:3--dest:2)
State 7 Depth = 6
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop = 0 rate = 60.0) (prevHop = 2 rate = 60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: source = 3
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:2--dest:1)
State 8 Depth = 7
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop = 0 rate = 60.0) (prevHop = 2 rate = 60.0); Data Cache: source = 2
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: source = 3
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:0) DATA(src:1--dest:2)
State 9 Depth = 8
        Node 0 Interest Cache: Empty; Data Cache: source = 1
        Node 1 Interest Cache: (prevHop = 0 rate = 60.0) (prevHop = 2 rate = 60.0); Data Cache: source = 2
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: source = 3
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:2) POSREINFORCE(src:0--dest:1)
State 10 Depth = 9
        Node 0 Interest Cache: Empty; Data Cache: source = 1
        Node 1 Interest Cache: (prevHop = 0 rate = 5.0) (prevHop = 2 rate = 60.0); Data Cache: source = 2
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: source = 3
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:2) POSREINFORCE(src:1--dest:2)
```

Figure C.1: Directed diffusion case study: Trace of counterexample 1 obtained using BFS-AC (continued next page).

```
State 11 Depth = 10
        Node 0 Interest Cache: Empty; Data Cache: source = 1
        Node 1 Interest Cache: (prevHop = 0 rate = 5.0) (prevHop = 2 rate = 60.0); Data Cache: source = 2
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:2) POSREINFORCE(src:1--dest:2)
State 12 Depth = 11
        Node 0 Interest Cache: Empty; Data Cache: source = 1
        Node 1 Interest Cache: (prevHop = 0 rate = 5.0) (prevHop = 2 rate = 60.0); Data Cache: source = 2
        Node 2 Interest Cache: (prevHop = 1 rate = 60.0); Data Cache: source = 1
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) POSREINFORCE(src:1--dest:2) DATA(src:2--dest:1)
State 13 Depth = 12
        Node 0 Interest Cache: Empty; Data Cache: source = 1
        Node 1 Interest Cache: (prevHop = 0 rate = 5.0) (prevHop = 2 rate = 60.0); Data Cache: source = 2
        Node 2 Interest Cache: (prevHop = 1 rate = 5.0); Data Cache: source = 1
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:2--dest:1) POSREINFORCE(src:2--dest:1)
State 14 Depth = 13
        Node 0 Interest Cache: Empty; Data Cache: source = 1
        Node 1 Interest Cache: (prevHop = 0 rate = 5.0) (prevHop = 2 rate = 5.0); Data Cache: source = 2
        Node 2 Interest Cache: (prevHop = 1 rate = 5.0); Data Cache: source = 1
        Node 3 Interest Cache: (prevHop = 2 rate = 60.0); Data Cache: source = 3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:2--dest:1) POSREINFORCE(src:1--dest:2)
```

Figure C.1: (cont.) Directed diffusion case study: Trace of counterexample 1 obtained using BFS-AC.

```
Counterexample
State 1 Depth = 0
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network Empty
State 2 Depth = 1
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:0--dest:1)
State 3 Depth = 2
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 4 Depth = 3
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) INTEREST(src:2--dest:3)
State 5 Depth = 4
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) INTEREST(src:3--dest:2) DATA(src:3--dest:2)
State 6 Depth = 5
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0) (prevHop=3 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2)
State 7 Depth = 6
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0) (prevHop=3 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2) INTEREST(src:0--dest:1)
State 8 Depth = 7
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0) (prevHop=3 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2) INTEREST(src:0--dest:1)
State 9 Depth = 8
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0) (prevHop=3 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2)
                INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 10 Depth = 9
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0) (prevHop=3 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) DATA(src:3--dest:2) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 11 Depth = 10
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0) (prevHop=3 rate=60.0); Data Cache: source=3
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
                DATA(src:2--dest:1) DATA(src:2--dest:3)
```

Figure C.2: Directed diffusion case study: Trace of counterexample 2 obtained using DD-1-BeFS-AC (continued next page).

```
State 12 Depth = 11
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=3 rate=60.0); Data Cache: source=3
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
                DATA(src:2--dest:1) DATA(src:2--dest:3)
State 13 Depth = 12
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
                DATA(src:2--dest:1) DATA(src:2--dest:3)
State 14 Depth = 13
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (prevHop=1 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:1)
                DATA(src:2--dest:3) INTEREST(src:2--dest:1) INTEREST(src:2--dest:3)
State 15 Depth = 14
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: source=2
        Node 2 Interest Cache: (prevHop=1 rate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:1--dest:0) DATA(src:1--dest:2)
State 16 Depth = 15
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: source=2
        Node 2 Interest Cache: (prevHop=1 rate=60.0); Data Cache: source=1
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:1--dest:0) DATA(src:2--dest:1)
State 17 Depth = 16
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (prevHop=0 rate=60.0) (prevHop=2 rate=60.0); Data Cache: source=2
        Node 2 Interest Cache: (prevHop=1 rate=60.0); Data Cache: source=1
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:0--dest:1)
State 18 Depth = 17
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (prevHop=0 rate=5.0) (prevHop=2 rate=60.0); Data Cache: source=2
        Node 2 Interest Cache: (prevHop=1 rate=60.0); Data Cache: source=1
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:1--dest:2)
State 19 Depth = 18
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (prevHop=0 rate=5.0) (prevHop=2 rate=60.0); Data Cache: source=2
        Node 2 Interest Cache: (prevHop=1 rate=5.0); Data Cache: source=1
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:2--dest:1)
State 20 Depth = 19
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (prevHop=0 rate=5.0) (prevHop=2 rate=5.0); Data Cache: source=2
        Node 2 Interest Cache: (prevHop=1 rate=5.0); Data Cache: source=1
        Node 3 Interest Cache: (prevHop=2 rate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:1--dest:2)
```

Figure C.2: (cont.) Directed diffusion case study: Trace of counterexample 2 obtained using DD-1-BeFS-AC.

References

[1] A. Aldini, R. Gorrieri, M. Roccetti, and M. Bernardo. Comparing the QoS of Internet audio mechanisms via formal methods. *ACM Transactions on Modeling and Computer Simulation*, 11(1):1–42, January 2001.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proc. of ICSE 2005*.

[3] O. Balci. Quality assessment, verification, and validation of modeling and simulation applications. In *Proc. of the 2004 Winter Simulation Conference*.

[4] O. Balci. Verification, validation, and certification of modeling and simulation applications. In *Proc. of the 2003 Winter Simulation Conference*.

[5] O. Balci. Principles of simulation model validation, verification, and testing. *Transactions of the Society for Computer Simulation International*, 14(1):3–12, 1997.

[6] O. Balci. Verification, validation, and testing. In *The Handbook of Simulation*, pages 335–393. ed. J. Banks. New York, NY: John Wiley & Sons, 1998.

[7] O. Balci, R. E. Nance, J. D. Arthur, and W. F. Ormsby. Expanding our horizons in verification, validation, and accreditation research and practice. In *Proc. of the 2002 Winter Simulation Conference*.

[8] O. Balci and R. G. Sargent. Validation of simulation models via simultaneous confidence intervals. *American Journal of Mathematical and Management Sciences*, 4(3-4):375–406, 1984.

[9] J. Banks, J. S. Carson II, B. L. Nelson, and D. M. Nicol. *Discrete-event System Simulation*. Prentice Hall, Inc., 2005.

[10] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[11] M. Bernardo. An algebra-based method to associate rewards with EMPA terms. In *Proc. of ICALP 1997*.

[12] M. Bernardo, W. R. Cleaveland, S. T. Sims, and W. J. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In *Proc. of IFIP FORTE/PSTV 1998*.

[13] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, July 1998.

[14] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, February 2002.

[15] K. Bhargavan, C. A. Gunter, and D. Obradovic. Fault origin adjudication. In *Proc. of the Workshop on Formal Methods in Software Practice, August 2000.*

[16] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538–576, July 2002.

[17] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS 1999.*

[18] A. W. Brown, editor. *Component-Based Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[19] F. Calí, M. Conti, and E. Gregori. Dynamic tuning of the IEEE 802.11 protocol to achieve a theoretical throughput limit. *IEEE/ACM Transactions on Networking*, 8(6):785–799, December 2000.

[20] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of ICSE 2003.*

[21] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.3 manual. January 2007.

[23] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In *Proc. of CAV 1996.*

[24] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Proc. of CAV 2005.*

[25] J. Corbett, M. Dwyer, J. Hatcliff, C. Păsăreanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite state models from Java source code. In *Proc. of ICSE 2000.*

[26] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *Proc. of ICFEM 2006,* S*pringer*-V*erlag* LNCS *4260.*

[27] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proc. of IEEE ICCD 1992.*

[28] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon. Nest: A network simulation and prototyping testbed. *Communications of the ACM*, 33(10):64–74, October 1990.

[29] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. of ICSE 2007.*

[30] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. of ACM FSE 2006.*

[31] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2-3):247–267, March 2004.

[32] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *Proc. of WRLA 2002.*

[33] K. El-Fakih, N. Yevtushenko, and G. v. Bochmann. FSM-based incremental conformance testing methods. *IEEE Transactions on Software Engineering*, 30(7):425–436, 2004.

[34] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. of ICSE 2001.*

[35] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[36] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In *Proc. of CAV 2004*.

[37] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of ACM POPL 2005*.

[38] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of ACM POPL 1997*.

[39] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. of TACAS 2002*.

[40] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[41] A. Groce and W. Visser. Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4):260–276, August 2004.

[42] M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression test selection for Java software. In *Proc. of OOPSLA 2001*.

[43] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, 27(3):248–263, 2001.

[44] K. Havelund. Java PathFinder, A translator from Java to Promela. In *Proc. of SPIN 1999*.

[45] K. Heljanko, T. Junttila, and T. Latvala. Incremental and complete bounded model checking for full PLTL. In *Proc. of CAV 2005*.

[46] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A.A. Sanvido. Extreme model checking. In *Proceedings of the International Symposium on Verification: Theory and Practice*, 2003.

[47] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[48] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. of ACM MobiCom 2000*.

[49] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proc. of the IEEE International Conference on Automated Software Engineering (ASE 2001)*.

[50] J-Sim. http://www.j-sim.org/.

[51] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.

[52] Java PathFinder (JPF). http://javapathfinder.sourceforge.net/.

[53] S. K. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, S. Zabele, J. Kurose, and D. Towsley. Scalable fair reliable multicast using active services. *IEEE Network Magazine*, 14(1):48–57, January-February 2000.

[54] S. Keshav. Real : A network simulator. Technical Report 88/472, University of California, Berkeley, December 1988.

[55] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proc. of ECRTS 1999*.

[56] D. E. Knuth. *The art of computer programming, volume 2: seminumerical algorithms*. Addison-Wesley, 1998.

[57] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. of IEEE/ACM ICSE 2008*.

[58] A. M. Law. How to build valid and credible simulation models. In *Proc. of the 2006 Winter Simulation Conference*.

[59] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. of ICSE 2003*.

[60] J.A. Makowsky and E.V. Ravve. Incremental model checking for fixed point properties on decomposable structures. In *Proc. of MFCS 1995*.

[61] J. Martens, K. Pauwels, and F. Put. A neural network approach to the validation of simulation models. In *Proc. of the 2006 Winter Simulation Conference*.

[62] J. Martens, F. Put, and E. Kerre. A fuzzy set theoretic approach to validate simulation models. *ACM Transactions on Modeling and Computer Simulation*, 16(4):375–398, October 2006.

[63] Maude. http://maude.cs.uiuc.edu.

[64] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[65] V. Misra, W. Gong, and D. Towsley. Stochastic differential equation modeling and analysis of TCP windowsize behavior. In *Proc. of Performance 1999*, October 1999.

[66] M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In *Proc. of SPIN 2005*.

[67] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proc. of NSDI 2004*.

[68] M. Musuvathi, D. Y.W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. of OSDI 2002*.

[69] P. Ölveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In *Proc. of FASE 2001*.

[70] P. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *Proc. of FASE 2004*.

[71] P. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *Proc. of International Workshop on Parallel and Distributed Real-Time Systems 2006, held in conjunction with IEEE IPDPS 2006*.

[72] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of ESEC/FSE 2003*.

[73] A. Orso, M. J. Harrold, D. S. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metacontent to support the regression testing of component-based software. In *Proc. of ICSM 2001*.

[74] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*.

[75] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. of ACM SIGCOMM 1998*, September 1998.

[76] D. Y.W. Park, U. Stern, J. U. Skakkebæk, and D. L. Dill. Java model checking. In *Proc. of IEEE ASE 2000*.

[77] C. Perkins, E. Royer, and S. Das. Ad hoc on demand distance vector (aodv) routing. IETF Draft, January 2002.

[78] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Proc. of IEEE WMCSA 1999*.

[79] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. of OOPSLA 2004*.

[80] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[81] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[82] H. Rueß and L. de Moura. From simulation to verification (and back). In *Proc. of the 2003 Winter Simulation Conference*.

[83] M. Saaltink. The Z/EVES system. In *ZUM 1997: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, 1997.

[84] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. of ASE 2005*.

[85] A. K. Saha, K. To, S. PalChaudhuri, S. Du, and D. B. Johnson. Physical implementation of ad hoc network routing protocols using unmodified ns-2 models (poster). In *Proc. of ACM MobiCom 2004*.

[86] R. G. Sargent. Verification and validation of simulation models. In *Proc. of the 2005 Winter Simulation Conference*.

[87] S. Shakkottai and R. Srikant. How good are deterministic fluid models of internet congestion control? In *Proc. of IEEE INFOCOM 2002*, June 2002.

[88] A. Sobeih, W.-P. Chen, J. C. Hou, L.-C. Kung, N. Li, H. Lim, H.-Y. Tyan, and H. Zhang. J-Sim: A simulation environment for wireless sensor networks. In *Proc. of the Annual Simulation Symposium (ANSS 2005), part of the 2005 Spring Simulation Multiconference (SpringSim 2005)*.

[89] A. Sobeih, W.-P. Chen, J. C. Hou, L.-C. Kung, N. Li, H. Lim, H.-Y. Tyan, and H. Zhang. J-Sim: A simulation and emulation environment for wireless sensor networks. *IEEE Wireless Communications Magazine*, 13(4):104–119, August 2006.

[90] A. Sobeih, M. d'Amorim, M. Viswanathan, D. Marinov, and J. C. Hou. Verification of simulation models of network protocols using state space exploration and protocol-specific properties. *Submitted for publication*.

[91] A. Sobeih and J. C. Hou. A simulation framework for sensor networks in J-Sim. Technical Report UIUCDCS-R-2003-2386, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2003.

[92] A. Sobeih and S. Lauterburg. Incremental state space exploration in J-Sim. Technical Report UIUCDCS-R-2007-2898, Department of Computer Science, University of Illinois at Urbana-Champaign, September 2007.

[93] A. Sobeih, M. Viswanathan, and J. C. Hou. Check and Simulate: A case for incorporating model checking in network simulation. In *Proc. of ACM-IEEE MEMOCODE 2004*.

[94] A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In *Proc. of ICFEM 2005,* S*pringer-*V*erlag* LNCS *3785*.

[95] A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. J-Sim: An integrated environment for simulation and model checking of network protocols. In *Proc. of IEEE IPDPS 2007, NSF Next Generation Software Program Workshop*.

[96] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proc. of CAV 1994*.

[97] M. Spivey. *The* Z *Notation:* A *Reference Manual*. Prentice Hall International Series in Computer Science, 1992.

[98] W. J. Stewart. *Introduction to the Numerical Solution of* M*arkov Chains*. Princeton University Press, 1994.

[99] J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in FLAVERS. In *Proc. of ACM SIGSOFT 2004/FSE-12*.

[100] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International Inc., 1996.

[101] GloMoSim. http://pcl.cs.ucla.edu/projects/glomosim/.

[102] Java Native Interface: Programmer's Guide and Specification. Online book. http://java.sun.com/docs/books/jni/.

[103] The Network Simulator ns 2. http://www.isi.edu/nsnam/ns/.

[104] OPNET. http://www.opnet.com.

[105] Scalable Simulation Framework (SSF). http://www.ssfnet.org/homepage.html.

[106] Symbolic Analysis Laboratory. http://sal.csl.sri.com/.

[107] M. K. Traore. Analyzing static and temporal properties of simulation models. In *Proc. of the 2006 Winter Simulation Conference*.

[108] H.-Y. Tyan. *Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation*. PhD thesis, Department of Electrical Engineering, The Ohio State University, 2002.

[109] H.-Y. Tyan, A. Sobeih, and J. C. Hou. Design, realization, and evaluation of a component-based compositional network simulation environment. *Conditionally accepted for publication in SIMULATION: Transactions of The Society for Modeling and Simulation International. 2008*.

[110] H.-Y. Tyan, A. Sobeih, and J. C. Hou. Towards composable and extensible network simulation. In *Proc. of IEEE IPDPS 2005, NSF Next Generation Software Program Workshop*.

[111] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of IEEE ASE 2000*.

[112] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. of ACM ISSTA 2004*.

[113] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red-black trees using abstraction. In *Proc. of IEEE/ACM ASE 2005*.

[114] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *Proc. of ACM ISSTA 2006*.

[115] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. of ACM/IEE DAC 1998*.

[116] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, 2^{nd} edition, 2000.

[117] H. Zhang and J. C. Hou. Maintaining sensing coverage and connectivity in large sensor networks. *Wireless Ad Hoc and Sensor Networks: An International Journal*, 1(1-2):89–123, January 2005.

[118] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying regression test selection for COTS-based applications. In *Proc. of ICSE 2006*.

Author's Biography

Ahmed Adel Sobeih was born in Cairo, Egypt on July 21^{st}, 1976. He received his B.Eng. and M.Eng. degrees in Computer Engineering from Cairo University, Egypt in July 1999 and May 2002 respectively. From July 2003 to August 2004, Sobeih was a research assistant at the National Center for Supercomputing Applications (NCSA). From June 2005 to December 2005, he was an intern at the IBM T. J. Watson Research Center in Hawthorne, NY. Sobeih is a recipient of several honors and awards including IBM Invention Award (2007), Vodafone-U.S. Foundation Graduate Fellowship for two consecutive years (2006-2007 and 2007-2008), Richard T. Cheng Endowed Fellowship (2002-2003), Bachelor's Honors Award (1999) and Undergraduate Excellence Award for five consecutive years (1994-1999). Sobeih is a member of IEEE and a student member of ACM. In September 2007, he was elected to membership in The Honor Society of Phi Kappa Phi, the nation's oldest, largest and most selective all-discipline honor society. After graduation, Sobeih will join Microsoft Corporation in Redmond, WA.